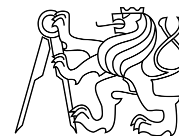
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ DIPLOMOVÉ PRÁCE

Název: Paralelní ešení soustavy lineárních rovnic a nerovnic pomocí ez v grafu
Student: Bc. Tomáš Ondrej
Vedoucí: Ing. Ivan Šime ek, Ph.D.
Studijní program: Informatika
Studijní obor: Systémové programování
Katedra: Katedra teoretické informatiky
Platnost zadání: Do konce letního semestru 2016/17

Pokyny pro vypracování

Nastudujte algoritmy ez , které jsou ur eny pro ešení soustav lineárních rovnic a nerovnic. Vyberte jeden vhodný algoritmus pro soustavy lineárních rovnic a jeden vhodný algoritmus pro soustavy lineárních nerovnic. Vybrané algoritmy implementujte v C/C++. Diskutujte teoretické možnosti paralelizace a prove te paralelní implementaci(e) samotných algoritm pomocí knihovny OpenMP. Srovnejte doby b hu s teoretickými p edpoklady a ostatními známými implementacemi t etích stran.

Seznam odborné literatury

Dodá vedoucí práce.

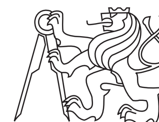
L.S.

doc. Ing. Jan Janoušek, Ph.D.
vedoucí katedry

prof. Ing. Pavel Tvrdík, CSc.
d kan

V Praze dne 25. listopadu 2015

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky



Diplomová práce

Paralelní řešení soustavy lineárních rovnic a nerovnic pomocí řezů v grafu

Bc. Tomáš Ondrej

Vedoucí práce: Ing. Ivan Šimeček, Ph.D.

4. května 2016

Poděkování

Nejprve bych chtěl poděkovat vedoucímu práce Ing. Ivanu Šimečkovi, Ph.D. za rady a připomínky k práci. Dále bych chtěl poděkovat své rodině a přátelům za podporu při studiu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 4. května 2016 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2016 Tomáš Ondrej. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Ondrej, Tomáš. *Paralelní řešení soustavy lineárních rovnic a nerovnic pomocí řezů v grafu.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Práce popisuje návrh, implementaci a měření rychlosti paralelní algoritmů pro řešení soustav lineárních rovnic a nerovnic. Algoritmy jsou implementovány v programovacích jazyce C/C++ s využitím knihovny OpenMP pro paralelizaci. Část práce se věnuje známým algoritmům, které tyto soustavy řeší.

Klíčová slova soustavy lineárních rovnic a nerovnic, lineární programování, paralelní algoritmy, C++, OpenMP, Metoda sdružených gradientů, Metoda větví a mezí, Simplexová metoda

Abstract

The thesis describes the design, implementation and measurement of the speed of parallel algorithms for solving systems of linear equations and inequalities. The algorithms are implemented in the programming language C/C++ using OpenMP library to parallelize. Part of the thesis deals with known algorithms that these systems solve.

Keywords systems of linear equations and inequalities, linear programming, parallel algorithms, C++, OpenMP, Conjugate gradient method, Branch and bound, Simplex method

Obsah

Úvod **1**

1 Úvod do problematiky **3**
1.1 Popis problémů . . . 3
1.2 Řešení soustav lineárních rovnic . . . 7
1.3 Řešení soustav lineárních nerovnic . . . 13
1.4 Měření kvality paralelních řešení . . . 21

2 Analýza a návrh řešení **25**
2.1 Specifikace cíle . . . 25
2.2 Podobné práce a řešiče . . . 26
2.3 Sekvenční řešení . . . 28
2.4 Paralelní řešení . . . 31

3 Realizace **43**
3.1 Knihovna OpenMP . . . 43
3.2 Implementace sekvenčních algoritmů . . . 46
3.3 Implementace paralelních algoritmů . . . 49

4 Měření **59**
4.1 Sekvenční algoritmy . . . 59
4.2 Paralelní algoritmy . . . 64
4.3 Srovnání se známými řešiči . . . 68

Závěr **71**

Literatura **73**

A Seznam použitých zkratek **77**

B Obsah přiloženého CD **79**

Seznam obrázků

1.1 Matice $A \in \mathbb{R}^{m,n}$ 3
1.2 Zápis soustavy lineárních rovnic 5
1.3 Zápis soustavy lineárních nerovnic 5
1.4 Osmistěn 6
1.5 Horní trojúhelníkový tvar rozšířené matice 7
1.6 Osmistěn obarvený podle účelové funkce 14
1.7 Simplexová tabulka s bazickým vektorem 16
1.8 Schéma matice D, vektoru z a vektoru q 18

2.1 Block-angular struktura 26
2.2 Plná matice 29
2.3 Kompresované řádky 30
2.4 Symetrické kompresované řádky 30
2.5 Paralelizace násobení matice s vektorem po řádcích 33
2.6 Paralelizace násobení matice s vektorem po sloupcích 34

3.1 Prohledávání stavové prostoru - čekání 54
3.2 Prohledávání stavové prostoru - častá komunikace 55

Seznam algoritmů

1.1 Metoda největšího spádu 11
1.2 Metoda sdružených směrů 12
1.3 Metoda sdružených gradientů 12
1.4 Simplexový algoritmus 16
1.5 Duální simplexový algoritmus 17
1.6 Metoda vnitřních bodů 19
2.1 Metoda sdružených gradientů - upravená verze 38
3.1 Násobení matice s vektorem - plná matice 51
3.2 Násobení matice s vektorem - kompresované řádky 51
3.3 Násobení matice s vektorem - symetrické kompresované řádky . 53

Seznam grafů

4.1 Závislost doby výpočtu na velikosti vstupní instance 60
4.2 Závislost doby výpočtu na hustotě vstupní instance 61
4.3 Závislost velikosti souboru na velikosti vstupní instance 61
4.4 Závislost doby výpočtu na poměru celočíselných proměnných . . . 62
4.5 Závislost doby výpočtu na počtu proměnných 63
4.6 Závislost doby výpočtu na počtu omezení 63
4.7 Závislost doby výpočtu na počtu vláken 64
4.8 Závislost zrychlení výpočtu na počtu vláken 65
4.9 Závislost zrychlení výpočtu na velikosti vstupní instance 65
4.10 Závislost doby výpočtu na počtu vláken 66
4.11 Závislost zrychlení výpočtu na počtu vláken 67
4.12 Závislost zrychlení výpočtu na počtu proměnných 67
4.13 Závislost doby výpočtu na hustotě vstupní instance 68
4.14 Závislost doby výpočtu na velikosti vstupní instance 69
4.15 Závislost doby výpočtu na poměru celočíselných proměnných . . . 70
4.16 Závislost doby výpočtu na počtu proměnných 70

Úvod

Na chvíli si představte, že vám někdo zadal následující úlohu: „Vlastníte pekárnu, která vyrábí n různých druhů pečiva a na výrobu každého z nich potřebujete m různých surovin. Pro každé pečivo máte přesně definováno, kolik které suroviny je potřeba k vyrobení jednoho kusu. Množství surovin je na skladě omezené. Dále víte, kolik utržíte za jeden kus daného pečiva. Kolik kusů kterého pečiva máte vyrobit, aby váš zisk byl maximální?“. Tento problém se nazývá „Optimální výrobní program“ a je jednou z typických úloh lineárního programování.

Problém výše se dá přepsat pomocí omezujících podmínek, které vedou k soustavě lineárních nerovnic, a optimalizačního kritéria, které v našem problému maximalizuje zisk z prodeje. Proto, když se mluví o lineárním programování, se jedná o řešení soustav lineárních nerovnic (popřípadě rovnic) s optimalizačním kritériem.

V praxi se pak lineární programování používá v úlohách jako jsou: optimální alokace zdrojů, plánování výroby, přiřazování práce procesorům, toky v sítích, problém obchodního cestujícího, aplikace v kryptografii, teorie her, data mining, výpočetní geometrie, statistika a další (převzato z [1]).

Podobně soustavy lineárních rovnic, kde většinou hledáme jedno řešení, mají v praxi veliké využití. Většinou se jedná o úlohy nelineární optimalizace: mechanická rovnováha, chemická rovnováha, regresní výpočty, minimalizace nákladů, minimalizace přepravní doby, maximalizace zisku a další (převzato z [2]).

Protože v reálných inženýrských úlohách se jedná o modely, které jsou trojrozměrné a bere se v úvahu více fyzikálních jevů, nebo se při plánování výroby počítá s více výrobky, odpovídající soustavy rovnic a nerovnic jsou poměrně obrovské (tyto soustavy mohou mít i milióny proměnných). Proto je důležité umět řešit tyto problémy rychle a efektivně a stejně tak pokračovat ve vylepšování stávajících algoritmů, neboť soustavy se stále a stále zvětšují.

Kapitola 1

Úvod do problematiky

Tato kapitola se zaobírá nutnou teorií k pochopení dané problematiky. Obsahuje definice a jejich vysvětlení, které se dále používají v práci. Kapitola nejdříve popisuje samotné problémy, dále existující algoritmy, které tyto problémy řeší, a v závěru kapitoly je sekce věnující se měřitelnosti kvality paralelizace. Jednotlivé sekce jsou rozděleny pro řešení soustav lineárních rovnic a pro řešení soustav lineárních nerovnic.

1.1 Popis problémů

1.1.1 Základní definice

V této subsekci budou popsány základní definice společné pro soustavy lineárních rovnic i nerovnic. Zmíněny budou jen ty definice, které se dále vyskytují v textu. Protože oba tyto systémy lze přepsat pomocí maticového zápisu (jak bude dále uvedeno), definice budou převážně ohledně matic a vektorů.

Definice 1.1.1 *Matice $A \in \mathbb{R}^{m,n}$ je soustava o m řádcích a n sloupcích, kde jednotlivé prvky a_{11}, a_{12} až a_{mn} nabývají reálných hodnot. Schéma takovéto matice je vidět na obr. 1.1.*

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

Obrázek 1.1: Matice $A \in \mathbb{R}^{m,n}$

Definice 1.1.2 *Vektor $x \in \mathbb{R}^n$ je n-tice uspořádaných prvků, kde jednotlivé prvky x_1, x_2 až x_n nabývají reálných hodnot. Pokud nebude dále řečeno jinak, bude se rozumět sloupcový vektor neboli vektor typu $\mathbb{R}^n$ bude odpovídat matici typu $\mathbb{R}^{n,1}$ a vektor x^T bude řádkově zapsaný vektor x (jedná se o transponovaný vektor).*

Definice 1.1.3 *Transponovaná matice A^T, respektive vektor x^T vznikne vzájemnou výměnou řádků a sloupců původní matice A, respektive vektoru x.*

Definice 1.1.4 *Hodnost h matice A je maximální počet lineárně nezávislých řádků matice A.*

Definice 1.1.5 *Čtvercová matice A je matice, pro kterou platí $A \in \mathbb{R}^{n,n}$. Jinak řečeno matice má stejný počet řádků jako sloupců.*

Definice 1.1.6 *Čtvercová matice A se nazývá regulární, jestliže je její hodnost rovna jejímu řádu (velikosti). V opačném případě, tj. když je její hodnost menší než její řád, se nazývá singulární (definice převzata z [3]).*

Definice 1.1.7 *Čtvercová matice A je symetrická, právě když všechny její prvky splňují $a_{ij} = a_{ji}$ neboli prvky jsou symetrické podle diagonály.*

Definice 1.1.8 *Čtvercová matice A je pozitivně definitní, právě když pro každý nenulový vektor $x \in \mathbb{R}^n$ platí $x^T A x > 0$.*

Definice 1.1.9 *Vlastní číslo λ matice A je číslo, pro které platí $Au = \lambda u$. Vektor u se pak nazývá vlastní vektor.*

Definice 1.1.10 *Euklidovská norma matice A, značena $\|A\|$, je číslo, pro které platí $\|A\| = \sqrt{\sum_{i=1}^{n} \sum_{j=1}^{m} a_{ij}^2}$.*

Definice 1.1.11 *Inverzní matice A^{-1} k matici A je matice, pro kterou platí $A \cdot A^{-1} = E$, kde E značí jednotkovou matici, pro kterou platí, že má na diagonále jedničky a mimo diagonálu nuly.*

1.1.2 Řešení soustav lineárních rovnic

Soustavu lineárních rovnic lze zapsat pomocí matice a vektorů jako $Ax = b$, kde i-tý řádek matice A a vektoru b obsahuje koeficienty proměnných a hodnotu levé strany i-té rovnice. Vektor x reprezentuje řešení této soustavy. Řešení této soustavy si můžeme představit jako průsečík nadrovin, které jsou dány jednotlivými rovnicemi. Schéma tohoto zápisu lze vidět na obr. 1.2.

Definice 1.1.12 *Matice A spolu s vektorem b tvoří tzv. rozšířenou matici k matici A a je typu $m, n+1$.*

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

Obrázek 1.2: Zápis soustavy lineárních rovnic

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \leq \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

Obrázek 1.3: Zápis soustavy lineárních nerovnic

Věta 1.1.1 (Frobeniova věta) *Nutnou a postačující podmínkou, aby soustava lineárních rovnic o n neznámých byla řešitelná, je, aby matice soustavy a rozšířená matice měly stejnou hodnost. Je-li tato podmínka splněna a označíme-li společnou hodnost obou matic h, pak je-li $h = n$, je daná soustava řešitelná jednoznačně; je-li $h < n$, je v dané soustavě možno hodnoty vhodných $n - h$ neznámých libovolně zvolit. Tímto způsobem dostaneme všechna řešení dané soustavy (věta převzata z [4]).*

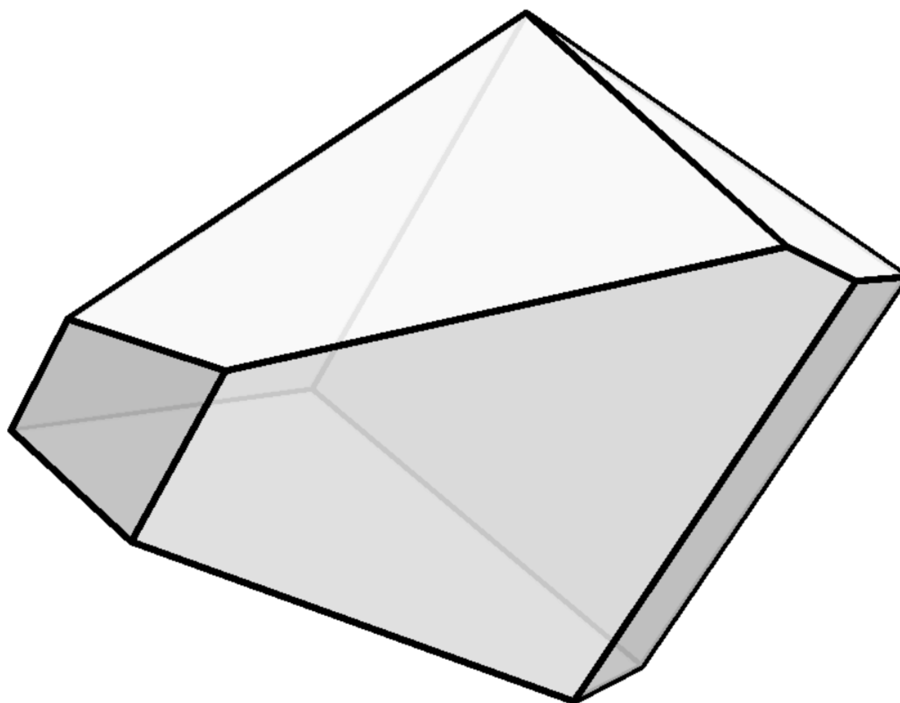
1.1.3 Řešení soustav lineárních nerovnic

Soustavu lineárních nerovnic, podobně jako soustavu lineárních rovnic, lze zapsat pomocí matice a vektorů jako $Ax \leq b$, kde i-tý řádek matice A a vektoru b obsahuje koeficienty proměnných a hodnotu levé strany i-té nerovnice. Vektor x reprezentuje řešení této soustavy. Schéma tohoto zápisu lze vidět na obr. 1.3.

Na rozdíl od řešení soustav lineárních rovnic, vznikne průsečíkem jednotlivých poloprostorů daných nerovnicemi polyedr (mnohostěn), který reprezentuje celý přípustný prostor možných řešení $\mathcal{F}$. Ukázku takového polyedru lze vidět na obr. 1.4.

Definice 1.1.13 *Prostor možných řešení $\mathcal{F}$ nazveme přípustným, pokud $\mathcal{F} \neq \emptyset$. V opačném případě ho nazveme nepřípustným (neobsahuje žádné možné řešení).*

Definice 1.1.14 *Prostor možných řešení $\mathcal{F}$ nazveme omezeným, pokud existuje hyperkoule S^n pro kterou platí $\mathcal{F} \subset S^n$ (existuje hyperkoule stejné di-*



Obrázek 1.4: Osmistěn

menze, do které lze celý prostor $\mathcal{F}$ uzavřít). V opačném případě ho nazveme neomezeným.

Protože možných řešení může existovat více, přidává se k soustavě lineárních nerovnic účelová funkce f a hledá se takové řešení x, pro které hodnota $f(x)$ nabývá největší, respektive nejmenší hodnoty. Takto zadefinovaný problém nazýváme lineárním programováním (dále jen LP). Soustavě lineárních rovnic pak říkáme omezení a funkci f nazveme optimalizačním kritériem.

Definice 1.1.15 *LP nazveme omezeným, pokud je prostor možných řešení $\mathcal{F}$ omezen ve směru účelové funkce f (optimální řešení neroste, respektive neklesá přes všechny meze). V opačném případě ho nazveme neomezeným.*

Obecný LP problém hledá optimum, kde všechny složky vektoru řešení x mohou nabývat reálných hodnot. Pokud se omezíme pouze na hledání řešení, jehož všechny složky jsou celočíselné, jedná se o celočíselné lineární programování značené ILP. Smíšené celočíselné lineární programování MILP je pak případ, kdy u některých složek chceme, aby byly celočíselné, a u některých stačí, aby byly reálné.

$$\left(\begin{array}{cccc|c} a_{11}^{(0)} & a_{12}^{(0)} & \cdots & a_{1n}^{(0)} & b_1^{(0)} \\ 0 & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} & b_2^{(1)} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a_{nn}^{(n-1)} & b_n^{(n-1)} \end{array}\right)$$

Obrázek 1.5: Horní trojúhelníkový tvar rozšířené matice

1.2 Řešení soustav lineárních rovnic

Většina metod popsána níže slouží k nalezení řešení soustav, které jsou jednoznačně řešitelné (mají právě jedno řešení). Proto, pokud nebude dále řečeno jinak, se bude počítat na vstupu s čtvercovou regulární maticí $A \in \mathbb{R}^{n,n}$ a vektorem pravé strany $b \in \mathbb{R}^n$. Hledané řešení soustavy pak bude $x \in \mathbb{R}^n$.

1.2.1 Přímé metody

Následující popis přímých metod byl převzat z [5]. Metody pro řešení soustav lineárních rovnic, které vedou k přesnému řešení (pokud se neberou v úvahu chyby numerického řešení) při konečném počtu výpočetních kroků, se označují jako metody přímé. Jejich základním rysem je eliminace neznámých. Pro plné matice bývají tyto metody nejefektivnější, při velkém počtu rovnic však může být výpočet omezen pamětí počítače.

1.2.1.1 Gaussova eliminace

Jedná se o nejstarší a nejznámější metodu pro řešení soustav lineárních rovnic. Jméno nese po jejím autorovi, slavném německém matematikovi a fyzikovi Carlu Friedrichu Gaussovi.

Metoda spočívá v postupné eliminaci neznámých v rozšířené soustavě rovnic. Tato eliminace se provádí pomocí řádkových úprav, které nemění vlastnosti soustavy (přičítání vhodného násobku jednoho řádku k druhému). Cílem těchto úprav je dosažení horního trojúhelníkovitého tvaru, jež lze vidět na obr. 1.5 (horní indexy značí počet přičtených násobků ostatních rovnic k dosažení dané eliminace).

Algoritmicky se eliminace provádí tak, že se vezme i-tý řádek matice a přičte se jeho násobek ke zbývajícím řádkům ($i + 1$ až n) tak, že na pozici i ve sloupci u těchto řádků vzniknou nuly. Násobkům pro jednotlivé řádky se říká multiplikátory. Z důvodu, že multiplikátory mohou vyjít rovny nule, a z důvodu vyšší numerické přesnosti se často provádí pivotáž. Pivotáž spočívá v přeuspořádání řádků a sloupců tak, aby koeficient v i-tém řádku a v i-tém sloupci v dané iteraci (koeficient, z kterého se vypočítávají multiplikátory) byl v absolutní hodnotě největší.

Po dosažení horního trojúhelníkovitého tvaru lze řešení snadno získat pomocí zpětné substituce. Zpětná substituce spočívá v postupném vyjadřování hodnot zespoda nahoru. Hodnota v posledním řádku je přímo vidět z matice, tedy $x_n = b_n^{(n-1)}/a_{nn}^{(n-1)}$. Tato hodnota se pak vezme a přímo se dosadí do rovnice na řádku $n-1$ a získá se tak hodnota x_{n-1}. Takto se dosazuje až k prvnímu řádku, kde se získají všechny hodnoty výsledného řešení.

Výpočetní složitost Gaussovy eliminace je $O(n^3)$.

1.2.1.2 LU dekompozice

Autorem této metody je významný britský matematik, logik, krypto-analytik a zakladatel moderní informatiky Alan Turing. Metoda LU dekompozice vychází z Gaussovy eliminace a její hlavním principem je rozložit matici A na dolní trojúhelníkovou matici L (lower) a horní trojúhelníkovou matici U (upper) tak, aby platilo $A = LU$ (vynásobením těchto dvou trojúhelníkových matic vznikne původní matice).

Poté můžeme vztah $Ax = b$ přepsat jako $LUx = b$. Dále můžeme označit $Ux = z$ a potom platí, že $LUx = b$ právě tehdy když $Lz = b$ a $Ux = z$. Takto rozloženou soustavu řešíme tak, že nejprve vyřešíme soustavu $Lz = b$ dopřednou substitucí a potom dosadíme z do pravé strany soustavy $Ux = z$, kterou řešíme zpětnou substitucí (převzato z [6]).

Metoda je vhodná pro řešení soustav lineárních rovnic, kde matice A zůstává stále stejná, ale vektor pravých stran b se mění. V takovém případě stačí matici A rozložit pouze jednou a akorát měnit vektor b podle konkrétních problémů, jež je mnohem rychlejší než pokaždé počítat řešení pomocí Gaussovy eliminace.

Výpočetní složitost LU dekompozice je opět $O(n^3)$.

1.2.1.3 Choleského dekompozice

Choleského dekompozice je modifikací předchozí LU dekompozice, kde matice A je navíc symetrická a positivně definitní. Pokud matice A splňuje tyto podmínky, lze ji opět rozdělit na součin dvou trojúhelníkových matic ale s tím rozdílem, že jedna je transpozicí té druhé. Platí pak $A = LL^T$.

Metoda dále pokračuje stejně jako LU dekompozice a má i stejné využití. Výhodou oproti LU dekompozici je menší paměťová náročnost (stačí si pamatovat jenom jednu trojúhelníkovou matici) a numerická stabilita výpočtu (chyba řešení způsobena danou aritmetikou roste nejvýše lineárně s počtem iterací).

1.2.2 Iterační metody

Iterační metody se od přímých liší tím, že nevypočítávají dané řešení najednou, ale postupně konstruují řešení, které se víc a víc blíží hledanému řešení. Jinak řečeno, každou iterací konvergují k řešení, které se od skutečného řešení liší

menší a menší chybou. Obecně lze zapsat, že pro iterační metodu danou funkcí f platí, že $x^{i+1} = f(x^i, x^{i-1}, \ldots, x^0)$ a $\lim_{i\to\infty} x^i = x$, kde i je číslo iterace a vektor x je skutečné hledané řešení.

Dalším společným znakem iteračních metod je, že uvnitř funkce f pro výpočet další iterace používají tzv. iterační matice. Tyto matice jsou zpravidla rozkladem či jinou úpravou původní matice soustavy a podle toho, jestli jsou pro každou iteraci stejné nebo ne, mluvíme o tzv. stacionárních metodách nebo nestacionárních metodách.

Dále je potřeba určit počáteční aproximaci x^0 a podmínku zastavení. Pokud nejsou známy žádné bližší informace o hledaném řešení, většinou se jako počáteční aproximace volí nulový vektor (stejně tak může být i libovolně jiný). Jako podmínka zastavení se bere dosažení požadované přesnosti nebo počtu iterací.

Výhodou těchto metod oproti přímým je, že přibližné řešení je k dispozici po celou dobu běhu výpočtu. Nevýhodou je, že přesné řešení nemusí být nalezeno nikdy (algoritmus se nemusí zastavit).

1.2.2.1 Jacobiova metoda

Tato metoda je založena na rozkladu matice A na součet diagonální matice D, dolní trojúhelníkové matice L a horní trojúhelníkové matice U, tedy $A = D + L + U$ (někdy se $L + U$ uvádí jako jedna matice R, reprezentující zbytek od diagonály).

Pokud dosadíme do původní soustavy lineárních rovnic, dostaneme $(D + L + U)x = b$. Toto se dá dále přepsat jako $Dx + (L + U)x = b$ a následně jako $Dx = b - (L + U)x$. Z poslední rovnice pak můžeme vyjádřit x jako $x = D^{-1}[b-(L+U)x]$ a tuto rovnici pak použijeme v rámci jedné iterace jako $x^{i+1} = D^{-1}[b - (L + U)x^i]$.

Protože se žádná z matic D^{-1}, L ani U v průběhu výpočtu nemění, jedná se o stacionární metodu a protože lze jednotlivé složky vektoru řešení x^{i+1} počítat na sobě nezávisle, lze tyto výpočty počítat paralelně.

1.2.2.2 Gauss-Seidelova metoda

Podobně jako Jacobiova metoda využívá rozkladu matice A na matice D, L a U. Liší se tím, že při úpravě rozdělující matice zůstane matice L u matice D, respektive $(D + L)x + (U)x = b$. Toto má za následek, že výsledná rovnice pro iteraci je $x^{i+1} = (D + L)^{-1}[b - Ux^i]$.

Tato malá změna má za důsledek, že vypočtené složky vektoru řešení x^{i+1} se ihned používají k výpočtu zbylých složek. Díky této změně Gauss-Seidelova metoda konverguje k přesnému řešení rychleji, ale nedá se počítat paralelně tak jako Jacobiova metoda.

1.2.2.3 Super-relaxační metoda

Následující popis metody byl částečně převzat z [7]. Super-relaxační metoda je úpravou Gauss-Seidelovy metody za účelem rychlejší konvergence. Zapíšeme-li vztah přibližných řešení mezi následujícími iteracemi jako $x^{i+1} = x^i + \Delta^i$, kde Δ^i reprezentuje změnu oproti stávajícímu řešení, pak pro Super-relaxační metodu platí $x^{i+1} = x^i + \omega\Delta^i$, kde ω je tzv. relaxační faktor.

Relaxační faktor ω se většinou volí z intervalu $(0, 2)$. Pro $\omega < 1$ se konvergence metody zpomalí, ale může vést k větší stabilitě metody. Pro $\omega = 1$ metoda odpovídá Gauss-Seidelově metodě a pro $\omega > 1$ se metoda zrychlí. Ostatní vlastnosti zůstávají jako u Gauss-Seidelovy metody.

1.2.3 Gradientní metody

Gradientní metody stejně jako iterační postupně konstruují řešení, které konverguje k přesnému řešení. Dokonce jsou známy takové metody, které v konečném počtu kroků dokonvergují k přesnému řešení. Hlavním rozdílem oproti iteračním metodám je takový, že kromě toho, aby matice A byla čtvercová a regulární, tak musí být ještě symetrická a pozitivně definitní.

Tyto vlastnosti umožňují převést problém hledání řešení soustavy lineárních rovnic $Ax = b$ na hledání minima kvadratické formy $\frac{1}{2}x^T Ax - x^T b + c$. Protože hodnota c nemá vliv na polohu minima (po první derivaci vypadne), můžeme ji z kvadratické formy odstranit a počítat dále jen s $\frac{1}{2}x^T Ax - x^T b$ (částečně převzato z [8]).

Metody se nazývají gradientní, protože v každém kroku při hledání nové aproximace přesného řešení x vybírají směr největšího poklesu kvadratické formy (gradient je v obecném smyslu slova směr růstu).

1.2.3.1 Metoda největšího spádu

Metoda největšího spádu v každém kroku vybere směr největšího spádu v daném bodě (bodem myslíme aktuální aproximaci řešení x) a přesune se do nového bodu, které leží v daném směru a jeho hodnota je minimální (minimální z bodů v daném směru). Takto iteruje dokud nenastane zastavovací podmínka.

Gradient, směr největšího růstu v bodě, kvadratické formy $\frac{1}{2}x^T Ax - x^T b$ je roven její derivaci, jež se rovná $Ax - b$. Proto reziduum r, odpovídající směru největšího poklesu, je rovno $b - Ax$. Přesun do nového bodu (nová aproximace) v k-té iteraci má poté tvar $x_{k+1} = x_k + \alpha_k r_k$, kde α_k reprezentuje délku kroku v daném směru v k-té iteraci. Dosazením tohoto výrazu do původní kvadratické formy a minimalizací podle α_k dostane $\alpha_k = \frac{r_k^T r_k}{r_k^T A r_k}$ a nakonec ze vztahu pro původní reziduum dostaneme $r_{k+1} = r_k - \alpha_k A r_k$.

Algoritmus konverguje z libovolného počátečního řešení x_0 k přesnému řešení x, ale nemusí se zastavit po konečném počtu iterací. Proto se jako

zastavovací podmínka bere klesnutí velikosti rezidua r pod určitou mez ϵ. Pseudokód algoritmu je zachycen v alg. 1.1.

Algoritmus 1.1 Metoda největšího spádu

```
function METODANEJVETSIHOSPADU(A, b, x_0, ε)
    r_0 = b − A x_0
    k = 0

    while r_k^T r_k > ε do
        α_k = (r_k^T r_k) / (r_k^T A r_k)
        x_{k+1} = x_k + α_k r_k
        r_{k+1} = r_k − α_k A r_k
        k = k + 1

    return x_k
```

1.2.3.2 Metoda sdružených směrů

Metoda vycházející z metody největšího spádu, která ale navíc využívá faktu, že po sobě jdoucí směry r_k a r_{k+1}, jsou navzájem kolmé. U metody největšího spádu se mohou směry opakovat a z toho důvodu metoda nemusí nikdy skončit. Metoda sdružených směrů využívá každý směr právě jednou a protože v n-rozměrném prostoru existuje maximálně n nezávislých kolmých vektorů, metoda po n krocích dokonverguje k přesnému řešení.

Bohužel tyto vektory není možné najít, neboť se vypočítávají podle chyby v k-té iteraci (kdybychom tuto chybu znali, znali bychom rovnou i přesné řešení). Proto se používají A-konjugované (někdy jako A-ortogonální) vektory, pro které platí $d_i^T A d_j = 0$. Pro tyto vektory algoritmus opět skončí po n krocích, ale naštěstí se dají spočítat (např. pomocí Gram-Schmidt konjunktivního procesu) a pro nalezení řešení stačí vzít libovolnou n-tici A-konjugovaných vektorů (myšlenka převzata z [9]). Pseudokód algoritmu je zachycen v alg. 1.2.

1.2.3.3 Metoda sdružených gradientů

Metoda sdružených gradientů je vlastně metoda sdružených směrů, kde jednotlivé A-konjugované vektory odpovídají samotným reziduím a na jejich základě jsou pak postupně konstruovány směrové vektory udávající směr posunu. Toto umožňuje již zmíněná vlastnost, že jednotlivá rezidua jsou na sebe kolmá, a také to, že jsou nyní kolmá i na směrové vektory.

Na základě předchozích tvrzení vznikají dva nové vztahy pro směrové vektory, pro směr v další iteraci platí $d_{k+1} = r_{k+1} + \beta_k d_k$ a velikost kroku β platí $\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$. Pseudokód rozšířeného algoritmu je zachycen v alg. 1.3.

Algoritmus 1.2 Metoda sdružených směrů

function metodaSdruzenychSmeru$(A, b, x_0, \{d_i\}_{i=0}^{n-1}, \epsilon)$
$r_0 = b - Ax_0$
$k = 0$

while $r_k^T r_k > \epsilon$ **do**
$\alpha_k = \frac{d_k^T r_k}{d_k^T A d_k}$
$x_{k+1} = x_k + \alpha_k d_k$
$r_{k+1} = r_k - \alpha_k A d_k$
$k = k + 1$

return x_k

Algoritmus 1.3 Metoda sdružených gradientů

function metodaSdruzenychGradientu(A, b, x_0, ϵ)
$d_0 = r_0 = b - Ax_0$
$k = 0$

while $r_k^T r_k > \epsilon$ **do**
$\alpha_k = \frac{r_k^T r_k}{d_k^T A d_k}$
$x_{k+1} = x_k + \alpha_k d_k$
$r_{k+1} = r_k - \alpha_k A d_k$
$\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
$d_{k+1} = r_{k+1} + \beta_k d_k$
$k = k + 1$

return x_k

1.2.3.4 Metoda předpodmíněných sdružených gradientů

Jedná se o metodu sdružených gradientů, která se ale navíc snaží snížit číslo podmíněnosti matice A. Číslo podmíněnosti má vliv na rychlost konvergence k přesnému řešení (zmíněno v [4]).

Definice 1.2.1 *Číslo podmíněnosti matice A, značeno $cond(A)$, je číslo, pro které platí $cond(A) = \|A\| \cdot \|A^{-1}\|$. Pokud je matice A navíc pozitivně definitní, platí pro její číslo podmíněnosti $cond(A) = \frac{\lambda_{max}}{\lambda_{min}}$, kde λ_{max} je největší vlastní číslo matice A a λ_{min} je nejmenší vlastní číslo matice A .*

Před-podmiňování je výběr symetrické positivně definitní matice M, která lze snadno invertovat a pro kterou platí, že matice vzniklá na základě součinu

$M^{-1}A$ má menší číslo podmíněnosti. Výchozí soustava lineárních rovnic se pak dá přepsat na $M^{-1}Ax = M^{-1}b$.

Obecně neplatí, že součin dvou symetrických pozitivně definitních matic dá opět symetrickou pozitivně definitní matici, ale existuje celá řada metod sdružených gradientů s podmíněním, které toto zajišťují. Mezi takovéto metody patří např. Transformované předpodmíněné sdružené gradienty, Netransformované předpodmíněné sdružené gradienty nebo metoda, která využívá jednokrokový Jacobiho předpodmiňovač.

1.3 Řešení soustav lineárních nerovnic

Pro připomenutí, pokud se mluví o řešení soustav lineárních nerovnic, má se na mysli lineární programování. Lineární programování je optimalizační problém, kde je dána soustava omezení v podobě nerovnic $Ax \leq b$ a dále účelová funkce $f(x)$, přes kterou se hledá optimum. Pokud nebude dále řečeno jinak, účelová funkce bude maximalizovat řešení x a bude mít následující podobu $f(x) = c^T x$, kde transponovaný vektor c odpovídá cenám jednotlivých složek řešení x.

Věta 1.3.1 (von Neumann, silná věta o dualitě) *Nechť prostor $\mathcal{F}$ daný $Ax \leq b$ je neprázdný a účelová funkce f se rovná maximu z $c^T x$, kde $x \in \mathcal{F}$ a optimum neroste přes všechny meze. Potom prostor $\mathcal{G}$ daný $A^T y \geq c, y \geq 0$ s účelovou funkcí g rovnou minimu z $b^T y$, kde $y \in \mathcal{G}$, se nazývá duál a platí, že pokud existuje optimum primární úlohy x^*, pak také existuje optimum duální úlohy y^* a $c^T x^* = b^T y^*$.*

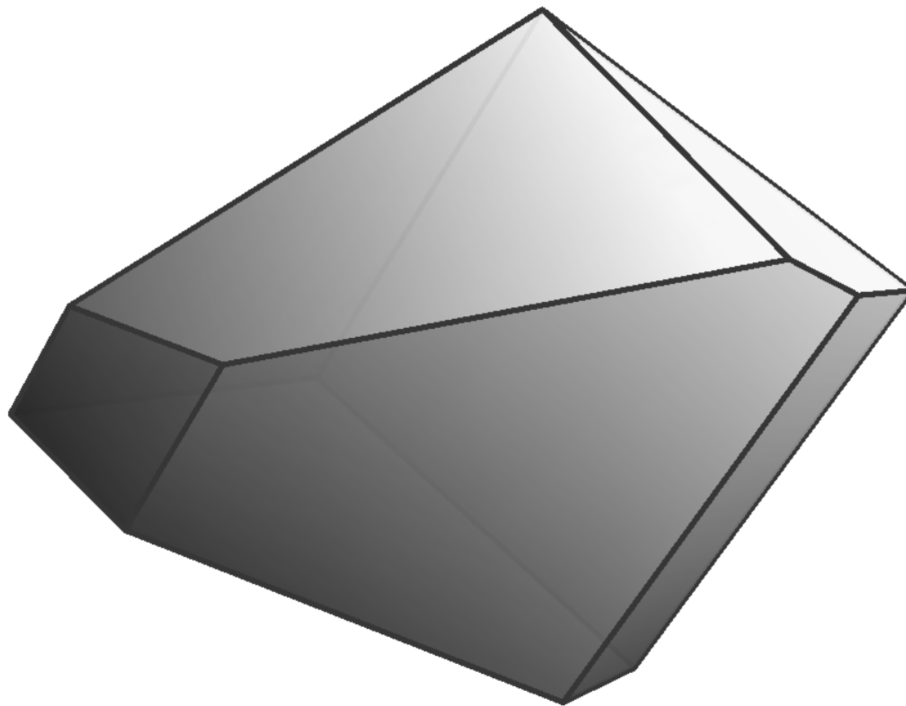
Dále jsou zmíněny nejznámější algoritmy řešící lineární programování, které jsou rozděleny do subsekcí lineární programování - LP, celočíselné lineární programování - ILP a smíšené celočíselné programování - MILP (tyto pojmy byly vysvětleny na konci subsekce 1.1.3).

1.3.1 Lineární programování

1.3.1.1 Simplexový algoritmus

„Simplexová metoda (Simplexový algoritmus) je iterativní způsob řešení problémů lineárního programování (lineární optimalizace) objevený americkým matematikem Georgem Dantzigem v roce 1947. Simplexová metoda postupuje od základního řešení, v každém svém kroku řešení pozmění takovým způsobem, aby hodnota účelové funkce byla vyšší než v kroku předchozím. Algoritmus končí, pokud řešení již nelze zlepšit (je optimální).“[10]

Následující text a myšlenky čerpají z [11]. Intuitivní verze fungování tohoto algoritmu se opírá o geometrii problému. Jak již bylo zmíněno v subsekci 1.1.3, soustava lineárních nerovnic vytvoří v prostoru polyedr, který reprezentuje všechna možná přípustná řešení. Pro jednodušší pochopení algoritmu se dále bude uvažovat polyedr, který je přípustný a omezený. Pokud



Obrázek 1.6: Osmistěn obarvený podle účelové funkce

by se všechny body tohoto polyedru obarvily podle hodnoty jejich účelové funkce (viz. obr. 1.6), bylo by hned zřejmé, že optimální hodnota se určitě nachází na povrchu tohoto polyedru a dokonce vždy alespoň jeden vrchol má optimální hodnotu. Na základě tohoto zjištění stačí „chodit“ po vrcholech ve směru růstu účelové funkce, dokud lze takovýto vrchol najít.

Tato verze se dá přepsat tak, že se vezme systém F, který odpovídá systému lineárních rovnic vzniklého z původního systému lineárních nerovnic nahrazením nerovností za rovnosti. Dále báze, značena B, pro kterou platí $B \subseteq \{1, 2, \ldots, m\}$, kde m je počet rovnic systému F. Poté F_B je subsystém lineárních rovnic tvořený právě těmi rovnicemi, jejichž indexy jsou obsaženy v množině B. Pokud má subsystém F_B, kde rovnice jsou lineárně nezávislé a $|B| = n$, jednoznačné řešení, pak toto řešení odpovídá vrcholu polyedru daným původními nerovnostmi. Z toho plyne, že každému vrcholu polyedru se dá přiřadit báze B, respektive uspořádaná n-tice čísel, která odpovídá indexům rovnic F, které tento vrchol splňuje.

Myšlenka algoritmu je poté taková, že na vstupu je dán výchozí vrchol v_0 polyederu daný bází B, dále jen v_B. Sousední vrcholy $v_{B'}$, jsou takové vrcholy, do kterých lze přejít z vrcholu v_B po jedné hraně. Proto z báze B se odstraní

index i odpovídající hraně, po které se přechází, a přidá se nový index j, který není obsažen v samotné bázi, neboli platí $B' = (B \setminus \{i\}) \cup j$. Tímto způsobem mohou vzniknout i báze, které neodpovídají vrcholům polyederu, proto se s nimi dále nepracuje. Ze sousedních uzlů se vybere ten uzel, pro který účelová funkce f vzroste nejvíce. Tento proces se iteruje dokud je možné přejít do uzlu, pro který účelová funkce f vzroste.

Speciálním případem jsou degenerované polyedry, kde jednomu vrcholu může odpovídat více bází. Z toho důvodu se buduje speciální strom, kde potomky aktuálního uzlu jsou všechny ostatní báze, které také odpovídají danému uzlu. V dalších iteracích je pak potřeba dát si pozor a nepřidávat do stromu báze, které se tam již vyskytují, aby nedošlo k zacyklení. Algoritmus skončí, až vyzkouší všechny možnosti ve stromě a zjistí, že opravdu nelze hodnotu účelové funkce f zvýšit.

Finální podoba simplexového algoritmu, která se umí vypořádat i s nepřípustným a neomezeným prostorem a nepotřebuje na vstup vrchol, pracuje ve dvou fázích, proto se jí také někdy říká dvoufázový algoritmus. V obou fázích dochází k přidání proměnných a omezení do nerovností tak, aby žádné z proměnných nemohly být záporné. V první fázi se polyedr transformuje tak, že leží v prvním ortantu (do části prostoru, kde všechny hodnoty na osách nabývají kladných hodnot) a je ohraničen zespodu souřadnými osami. Takto transformovaný polyedr má určitě vrchol v počátku souřadnicovém systému neboli všechny proměnné nabývají nulových hodnot. Pustí se simplexový algoritmus z tohoto vrcholu a pokud je nalezené optimum rovné nule, vrátí se vrchol tohoto optima a prostor je přípustný. Pokud se optimum nerovná nule, prostor je nepřípustný a algoritmus končí. V druhé fázi se vezme vrchol nalezený první fází a opět se pouští simplexový algoritmus, který vrátí optimum nebo konstatuje, že optimální hodnota roste přes všechny meze. Nalezené optimum je optimum původní úlohy.

V praxi se pak zavádí simplexová tabulka, dále značena jako T, nad kterou se provádí jednotlivé operace. Tato tabulka T vznikne přidáním proměnných x', aby platilo $Ax+Ex' = b$, kde E je jednotková matice, $x_i \geq 0$ a $x'_j \geq 0$. Dále je potřeba cenový vektor c rozšířit tak, že pro nové proměnné x' obsahuje samé nuly. Vedle tabulky se dále drží vektor B, který pro každý řádek i tabulky T drží informaci o jeho bazické proměnné (tato proměnná nabývá hodnoty T_{i0}, respektive aktuální hodnoty vektoru b). Tento vektor B je na počátku vyplněn přidanými proměnnými x'. Tuto tabulku T spolu s bazickým vektorem B lze vidět na obr. 1.7.

Výsledný algoritmus nad tabulkou je poměrně jednoduchý, jedná se o opakovaný výběr řádku splňující dané podmínky pro danou iteraci a přičtení jeho násobku k ostatním řádkům. Pseudokód práce se simplexovou tabulkou je zachycen v alg. 1.4. V praxi se dále používá ještě Duální simplexový algoritmus, který vychází z věty o dualitě a pracuje se simplexovou tabulkou takřka stejně. Většinou se používá v dalších krocích po Simplexovým algoritmu. Pseudokód práce Duálního simplexového algoritmu se simplexovou tabulkou je zachycen

B	T	x_1	x_2	$\cdots$	x_n	x'_1	x'_2	$\cdots$	x'_m
$-$	opt	c_1	c_2	$\cdots$	c_n	0	0	$\cdots$	0
x'_1	b_1	a_{11}	a_{12}	$\cdots$	a_{1n}	1	0	$\cdots$	0
x'_2	b_2	a_{21}	a_{22}	$\cdots$	a_{2n}	0	1	$\cdots$	0
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
x'_m	b_m	a_{m1}	a_{m2}	$\cdots$	a_{mn}	0	0	$\cdots$	1

Obrázek 1.7: Simplexová tabulka s bazickým vektorem

v alg. 1.5.

Algoritmus je velmi rychlý a často se používá a to i přesto, že má exponenciální složitost. Dodnes vědci zkoumají, jak je možné, že algoritmus v praxi běží tak rychle.

Algoritmus 1.4 Simplexový algoritmus

function SIMPLEX(T, B)
  while dokud existuje i, pro který platí $T_{0i} \leq 0$ **do**
    klíčový sloupec i, pro který platí, že hodnota T_{0i} je minimální
    (pří nejednoznačnosti se bere nejnižší sloupec)
    klíčový prvek T_{ji}, pro který platí $T_{ji} > 0$ a $\frac{T_{j0}}{T_{ji}}$ je minimální
    (pokud není žádný takový prvek, problém je neomezený)
    vynásobení řádku j hodnotou $\frac{1}{T_{ji}}$ (bude platit $T_{ji} = 1$)
    přičtení násobku řádku j k ostatním řádkům k, aby platilo $T_{ki} = 0$
    $B_j = i$ (přiřazení indexu klíčového sloupce do báze B)

  konstrukce vektoru řešení x tak, že se vezme hodnota B_j jako index do vektoru x, kam se přiřadí hodnota T_{j0} neboli $x_{B_j} = T_{j0}$ (berou se jen řádky j, pro které platí $B_j \leq n$, kde n je velikost vektoru x)

  return optimální řešení x a hodnota optima T_{00}

1.3.1.2 Elipsoidový algoritmus

Elipsoidový algoritmus je dalším algoritmem, který řeší lineární programování. Opírá se o geometrii problému a o větu o dualitě. Důsledek věty o dualitě je ten, že řešit optimalizační verzi úlohy je ekvivalentní k řešení rozhodovací verze této úlohy. Na základě tohoto důsledku elipsoidový algoritmus řeší pouze rozhodovací verzi úlohy (zda existuje řešení), respektive hledá vnitřní bod

Algoritmus 1.5 Duální simplexový algoritmus

```
function DUALSIMPLEX(T, B)
    while dokud existuje j, pro který platí T_j0 ≤ 0 do
        klíčový řádek j, pro který platí, že hodnota T_j0 je minimální
        (pří nejednoznačnosti se bere nejnižší řádek)
        klíčový prvek T_ji, pro který platí T_ji < 0 a −T_0i/T_ji je minimální
        (pokud není žádný takový prvek, problém je neomezený)
        vynásobení řádku j hodnotou 1/T_ji (bude platit T_ji = 1)
        přičtení násobku řádku j k ostatním řádkům k, aby platilo T_ki = 0
        B_j = i (přiřazení indexu klíčového sloupce do báze B)

    konstrukce vektoru řešení x tak, že se vezme hodnota B_j jako index
    do vektoru x, kam se přiřadí hodnota T_j0 neboli x_{B_j} = T_j0 (berou se
    jen řádky j, pro které platí B_j ≤ n, kde n je velikost vektoru x)

    return optimální řešení x a hodnota optima T_00
```

polyedru. Tato metoda má dlouhou historii a zkoumala jí spousta lidí. První verze byla představena Naumem Z. Shorem.

Myšlenka algoritmu je taková, že na začátku se vezme dostatečně velký elipsoid, kterým se obalí celý polyedr. Pokud střed tohoto elipsoidu leží uvnitř polyedru, algoritmus nalezl řešení a končí. Pokud neleží, elipsoid se rozřízne skrz střed rovnoběžně s nesplňujícím omezením a ta část, která neobsahuje polyedr se zahodí. Ponechaná část se opět obalí elipsoidem a takto se iteruje, dokud objem elipsoidu neklesne pod mez, pro kterou je jisté, že přípustný prostor je prázdný.

Chačijanova verze tohoto algoritmu pracuje v polynomiálním čase, ale navíc řeší problém s aritmetikou, respektive se zaokrouhlováním. Toto má obrovský důsledek a to ten, že lineární programování spadá do třídy polynomiálně řešitelných algoritmů. Chačijanova věta, která říká, že $LP \in P$, byla uvedena roku 1979.

1.3.1.3 Metody vnitřních bodů

Metody vnitřních bodů, podobně jako předchozí dva algoritmy, staví na geometrii problému. Simplexový algoritmus se pohyboval po uzlech a hranách polyedru a elipsoidový algoritmus se pohyboval celou dobu mimo polyedr, dokud střed elipsoidu neležel uvnitř polyedru nebo dokud objem elipsoidu neklesl pod jistou mez. Metody vnitřních bodů se po celou dobu pohybují uvnitř polyedru a postupně konstruují z přípustného řešení řešení, které se více a více blíží optimálnímu, respektive se blíží k hranici polyedru, kde leží optimum.

Od roku 1984 byly publikovány doslova tisíce článků týkajících se metod

$$D = \begin{pmatrix} 0 & A & -b & 1-(A-b) \\ -A^T & 0 & c & 1-(-A^T+c) \\ b^T & -c^T & 0 & 1-(b^T-c^T) \\ (1-(A-b))^T & (1-(-A^T+c))^T & (1-(b^T-c^T))^T & 0 \end{pmatrix},$$

$$z = \begin{pmatrix} y \\ x \\ \kappa \\ \theta \end{pmatrix} \quad a \quad q = \begin{pmatrix} 0_{m\times 1} \\ 0_{n\times 1} \\ 0 \\ n+m+2 \end{pmatrix}, \quad kde \quad n = |x| \quad a \quad m = |y|$$

Obrázek 1.8: Schéma matice D, vektoru z a vektoru q

vnitřních bodů. Velmi zhruba se dají rozdělit na metody snižování potenciálu, metody sledování centrální cesty (ty se dnes zdají nejlepší pro praktické použití) a metody afinních transformací (částečně převzato z [12]). Dále bude popsána metoda centrální cesty.

Metoda centrální cesty na vstupu očekává LP problém daný omezeními $Ax \geq b$, $x_i \geq 0$ a účelovou funkcí $f(x) = c^T x$, kterou minimalizujeme. Od toho systému pomocí duálu přejdeme k systému $Ax - b\kappa \geq 0$, $-A^T y + c\kappa \geq 0$, $b^T y - c^T x \geq 0$, $x_i \geq 0$, $y_i \geq 0$ a $\kappa \geq 0$, jež se nazývá autoduální vnoření (je si samo sebou duálem). Hledá se takové řešení, pro které $\kappa > 0$.

Systém se dále modifikuje na $Dz \geq -q$, $z \geq 0$, aby podmínka vnitřního bodu byla splněna. Schéma matice D, vektoru z a vektoru q lze vidět na obr. 1.8. Jako „slack" nazveme funkci s, pro kterou platí $s(z) = Dz + q$. Pro řešení poté platí, že $z \geq 0$, $s(z) \geq 0$ a $z^T s(z) = 0$, jež odpovídá tzv. striktně komplementárnímu optimu (pokud je položka na i-té pozici u vektoru z kladná, tak i-tá položka u slacku je nulová a naopak).

Definice 1.3.1 *Polyedr $A_1 x = b_1$, $A_2 x \leq b_2$ splňuje podmínku vnitřního bodu (interior point condition, IPC), jestliže existuje a je znám bod x_0 takový, že $A_1 x_0 = b_1$ a $A_2 x_0 < b_2$ (definice převzata z [13]).*

Pro $z = 1$ platí, že je vnitřním bodem modifikovaného polyedru, ale nesplňuje podmínku striktní komplementarity, protože $z^T s(z) = 1$. Přepsáním této podmínky na $z^T s(z) = \mu 1$, hledání optimálního řešení odpovídá postupnému snižování μ (označováno jako μ-střed) z počáteční hodnoty $\mu^0 = 1$ směrem k nule. Všechny μ-středy poté tvoří centrální cestu, po které je metoda pojmenována.

Vyjádřením přechodu mezi řešeními z v k-té iteraci pomocí $\Delta^k z$ dostáváme vztah $z^{k+1} = z^k + \Delta^k z$. Poté dostáváme systém $D\Delta^k z - \Delta^k s = 0$, $z^k s^k + z^k \Delta^k s + s^k \Delta^k z + \Delta^k z \Delta^k s = \mu^k 1$, kde jediné neznámé jsou $\Delta^k z$ a $\Delta^k s$. Aby algoritmus dostatečně rychle konvergoval (stále byl polynomiální), je po-

třeba odstranit kvadratický člen $\Delta^k z \Delta^k s$. Zanedbání tohoto členu se nazývá Newtonovský krok.

V důsledku Newtonovskému kroku jednotlivá řešení neodpovídají přímo μ-středům, ale pouze jejich aproximacím, proto je třeba volit délku kroku Θ takovou, aby se aproximace neodchýlila natolik, že by nedokonvergovala k striktně komplementárnímu řešení. Proto se Θ volí rovno $\frac{1}{\sqrt{2(n+m+2)}}$. Pseudokód metody vnitřních bodů pomocí centrální cesty je zachycen v alg. 1.6.

Metody se dále dělí podle délky kroku na krátké a dlouhé. Krátké odpovídají výše popsanému algoritmu a dlouhé používají mnohem delší krok, aby algoritmus rychleji konvergoval. Ve většině případě se podaří trefit blízko dané aproximace i s velkým krokem, případně několika pokusy. Pokud se algoritmu využívající dlouhý krok nepodaří trefit blízko tak, aby mohl dále pokračovat, vrátí se krátkému kroku.

Metody vnitřních bodů vždy pracují pouze uvnitř polyedru, mohou se s libovolnou přesností nablížit optimu, ale nikdy ho nedosáhnou. Proto se na konci algoritmů používá nějaký heuristický krok, který skočí do optima, nebo se do cílového optima dostává pomocí Simplexového algoritmu (viz. 1.3.1.1).

Algoritmus 1.6 Metoda vnitřních bodů

function METODAVNITRNICHBODU($A, b, c, \epsilon, \Theta \in (0,1)$)
  vytvoření matice D, vektorů z a q z matice A, vektorů b a c
  $z^0 = 1$, $\mu^k = 1$

  while $q^T z^k \geq \epsilon$ **do**
    $\mu^{k+1} = (1-\Theta)\mu^k$
    $\Delta^k z$ je řešením $D\Delta^k z - \Delta^k s = 0$, $z^k s^k + z^k \Delta^k s + s^k \Delta^k z = \mu^k 1$
    $z^{k+1} = z^k + \Delta^k z$

  rozklad vektoru z^{k+1}
  return x

1.3.2 Celočíselné lineární programování

1.3.2.1 Výčtová metoda

Výčtová metoda je založená na tom, že vezme všechna možná řešení (pokud je dán omezený polyedr, tak počet celočíselných řešení je konečný počet) a postupně je projde a vybere z nich optimum. Na základě nerovností vzniká n-dimenzionální mřížka, která obaluje všechna možná celočíselná řešení. Tato mřížka ale obsahuje i nepřípustná řešení, která je nutno oddělit.

Tato metoda je vhodná pouze pro malé soustavy s omezeným počtem celočíselných proměnných. V praxi se příliš nepoužívá, uplatnění nachází pouze u ILP, kde proměnné nabývají pouze binárních hodnot. Potom je možné pro-

hledávání realizovat jako binární strom, což umožňuje redukovat počet nepřípustných stavů (převzato z [14]).

1.3.2.2 Metoda větví a mezí

Metoda větví a mezí pracuje tak, že na začátku zanedbá požadavky na celočíselnost a řeší problém jako LP. Dále se podívá na řešení, které je vráceno jako optimum LP, a rozdělí prohledávaný prostor podle jedné ze složky řešení, která není celočíselná. Označíme-li tuto složku řešení x jako x_k, poté se nám původní soustava nerovnic rozdvojí tak, že na jednu stranu (větev) přidáme nerovnici $x_k \leq \lfloor h(x_k) \rfloor$ a na druhou stranu nerovnici $x_k \geq \lfloor h(x_k) \rfloor + 1$, kde $h(x_k)$ je hodnota složky x_k v aktuálním řešení x.

Výpočet je rekurzivně volán na větve, dokud se nenalezne řešení, které je celočíselné. Hodnota účelové funkce tohoto celočíselného řešení je pak použita k prořezání ostatních větví a to tak, že pokud hodnota nějakého jiného řešení je menší (pokud maximalizujeme) nebo rovna tomuto celočíselnému řešení, větev se dále neprohledává (odřízne se).

Problém LP se většinou řeší pomocí Simplexového algoritmu (viz. 1.3.1.1), neboť po přidání nové nerovnice není potřeba pouštět algoritmus od znova, ale je možné navázat na předchozí výpočet LP řešením duální simplexové metody (zmíněno v [14]).

1.3.2.3 Metody sečných nadrovin

Metody sečných nadrovin podobně jako metoda větví a mezí používají Simplexový algoritmus (viz. 1.3.1.1), aby vypočítaly neceločíselné optimální řešení pro daný problém. Liší se tím, že se nevětví podle přidané podmínky, ale nově přidaná podmínka (sečná nadrovina) odřízne část prostoru tak, že neceločíselné optimální řešení již není v prohledávaném prostoru a všechna dosavadní přípustná celočíselná řešení v prostoru zůstávají. Takto se iteruje, dokud není nalezeno celočíselné optimální řešení (do simplexové tabulky přibude jeden řádek a jeden sloupec v každé iteraci). Mezi metody sečných nadrovin patří např. Dantzigovi řezy a Gomoryho řezy.

Dantzigovi řezy předpokládají na vstupu matici A a vektor b celočíselné. V každé iteraci k pak přidávají omezení ve tvaru $\sum_{x_i \in Q_k} x_i \geq 1$, kde Q_k jsou nebazické proměnné v iteraci k (proměnné, které byly postupně přidány a netvoří základní proměnné úlohy).

U Gomoryho řezů přidáváme podmínky ve tvaru $\sum_{x_i \in Q_k} f_i x_i - x_{n+1} = f_b$, kde f_i a f_b jsou celočíselné zbytky u koeficientů nebazických proměnných x_i a b pro řádek s největší desetinnou částí pro vektor b a x_{n+1} je nová nebazická proměnná (po nalezení optima LP v dané iteraci se vyber řádek v simplexové tabulce, kde ve sloupci odpovídajícímu vektoru b je největší desetinná část, a přidá se nové omezení takové, že se sepíšou desetinné části pro nebazické proměnné x_i a b v tomto řádku a přidá se nová nebazická proměnná).

1.3.3 Smíšené celočíselné lineární programování

Smíšené lineární programování se řeší metodami pro celočíselné lineární programování (kromě výčtové metody) s tím rozdílem, že u některých proměnných se prohlásí, že není potřeba, aby byly celočíselné, a při kontrole celočíselnosti nalezeného řešení pomocí lineárního programování (nejčastěji Simplexového algoritmu) se tyto proměnné přeskakují.

1.3.3.1 Metoda větví a řezů

Metoda větví a řezů, jak již název napovídá, kombinuje Metodu větví a mezí (viz. 1.3.2.2) dohromady s Metodou sečných nadrovin (viz. 1.3.2.3), konkrétně používá Gomoryho řezy. Metoda pracuje takřka stejně jako metoda větví a mezí, ale navíc si napříč stromem, který reprezentuje jednotlivé prohledané podprostory, posílá užitečné Gomoryho řezy, které urychlují finální algoritmus (na toto téma pojednává [15]).

1.4 Měření kvality paralelních řešení

Umět měřit kvalitu paralelních algoritmů, ale i sekvenčních, je důležité z toho důvodu, že v reálném světě procesory mají svojí cenu, někdy je potřeba získat řešení ihned a někdy se vyplatí si na řešení počkat za nižší náklady. Aby bylo možné porovnávat rozdílné verze paralelních algoritmů mezi sebou nebo oproti sekvenčnímu řešení, je potřeba zavést základní pojmy, které toto umožní. Následující text, definice složitostí, podsekce 1.4.1 a 1.4.2 čerpají z [16]. Podsekce 1.4.3 čerpá z [17].

Definice 1.4.1 *Časová složitost (doba výpočtu) sekvenčního algoritmu pro daný problém se značí $T(n)$, kde n je velikost vstupních dat. Počítá se nejhorší (nejpomalejší) možná doba běhu algoritmu na různých datech.*

Definice 1.4.2 *Spodní mez časové složitost sekvenčního algoritmu pro daný problém se značí $SL(n)$, kde n je velikost vstupních dat. Spodní mez odpovídá časové složitosti nejlepšímu (nejrychlejší) možnému sekvenčnímu algoritmu.*

Definice 1.4.3 *Horní mez časové složitost sekvenčního algoritmu pro daný problém se značí $SU(n)$, kde n je velikost vstupních dat. Horní mez odpovídá časové složitosti nejlepšímu (nejrychlejší) známému sekvenčnímu algoritmu.*

Definice 1.4.4 *Časová složitost (doba výpočtu) paralelního algoritmu pro daný problém se značí $T(n, p)$, kde n je velikost vstupních dat a p je počet procesorů. Za dobu výpočtu se považuje čas mezi začátkem algoritmu a okamžikem, kdy doběhne poslední procesor. Počítá se nejhorší (nejpomalejší) možná doba běhu algoritmu na různých datech.*

1.4.1 Zrychlení

Paralelní zrychlení je veličina udávající, jak moc je daný paralelní algoritmus rychlejší oproti sekvenčnímu algoritmu. V některých případech se toto porovnání bere oproti nejlepšímu známému sekvenčnímu řešení. Přirozeným cílem je dosáhnout lineárního zrychlení, respektive aby přidáním k procesorů, klesla doba výpočtu k-krát. Tohoto cíle je však těžké dosáhnout, neboť během výpočtu procesory potřebují spolu komunikovat a tak vzniká režie, kterou sekvenční algoritmus nemusí vůbec vykonávat.

Definice 1.4.5 *Paralelní zrychlení daného paralelního algoritmu pro daný problém se značí $S(n,p)$ a platí $S(n,p) = \frac{T(n)}{T(n,p)} \leq p$, kde n je velikost vstupních dat a p je počet procesorů.*

Někdy může docházet dokonce k tzv. superlineárnímu zrychlení, jež se označuje jako anomálie. Jedná se o zrychlení, pro které platí $S(n,p) > p$ (teoretická horní mez na zrychlení je počet procesorů neboli by mělo platit $S(n,p) \leq p$). K tomuto jevu může docházet v úlohách prohledávání stavového prostoru v důsledku jeho ořezávání (sekvenčně by se procesor mohl zanořit hluboko do nějaké větve a přitom by optimum, které by tuto větev ořízlo, mohlo ležet hned ve druhé větvi, kterou si v paralelní verzi vezme další procesor hned na začátku) nebo z hardwarových důvodů (typicky se sekvenční úloha nemusí vejít do paměti, ale při více výpočetních uzlech ano, protože je paměti mnohem více).

1.4.2 Škálovatelnost

Škálovatelnost je vlastnost paralelní algoritmů efektivně využít rostoucí počet procesorů nebo udržet efektivnost či dobrý čas při změnách velikosti vstupních dat a počtu procesorů. K tomu se používají izoefektivní funkce $\psi_1(p)$ a $\psi_2(n)$, které vyjadřují, jak musí velikost vstupních dat n růst s počtem procesorů p a naopak, aby se efektivnost neměnila.

Definice 1.4.6 *Paralelní efektivnost daného paralelního algoritmu pro daný problém se značí $E(n,p)$ a platí $E(n,p) = \frac{S(n,p)}{p}$, kde n je velikost vstupních dat a p je počet procesorů. Udává zrychlení na jeden procesor.*

Definice 1.4.7 *Funkce $\psi_1(p)$ je asymptoticky minimální funkce taková, že $\forall n_p = \Omega(\psi_1(p)) : E(n_p,p) \geq E_0$, kde E_0 je konstanta v intervalu $(0,1)$.*

Definice 1.4.8 *Funkce $\psi_2(n)$ je asymptoticky maximální funkce taková, že $\forall p_n = O(\psi_2(n)) : E(n,p_n) \geq E_0$, kde E_0 je konstanta v intervalu $(0,1)$.*

1.4.3 Amdahlův zákon

Amdahlův zákon vyjadřuje vztah pro zrychlení při vylepšení některé části počítače. Ekvivalentně pro algoritmy vyjadřuje vztah pro zrychlení, pokud dojde k paralelizaci jen některé části. Každý paralelní algoritmus se dá rozdělit na části, které se musí provádět sekvenčně (např. rozdělení práce na jednotlivé procesory nebo zpracování výsledků) a které se mohou provádět paralelně (např. výpočty nad jednotlivými daty).

Definice 1.4.9 *Amdahlův zákon říká, že* $S(n,p) \leq \frac{1}{T_s(n,p)+\frac{T_p(n,p)}{p}}$*, kde* $T_s(n,p)$ *je normovaný výpočet sekvenční části,* $T_p(n,p)$ *je normovaný výpočet paralelní části,* n *je velikost vstupních dat a* p *je počet procesorů (normovaný výpočet znamená, že platí* $T_s(n,p) + T_p(n,p) = 1$*).*

Kapitola 2

Analýza a návrh řešení

Na začátku této kapitoly jsou uvedeny cíle samotné práce a podobné práce na toto téma spolu s dostupnými řešiči. Dále se kapitola zaobírá výběrem vhodných algoritmů pro implementaci, analýzou možností spjatých s tímto výběrem a možnostmi paralelizace vybraných algoritmů. Jednotlivé sekce jsou rozděleny pro řešení soustav lineární rovnic a pro řešení soustav lineárních nerovnic.

2.1 Specifikace cíle

2.1.1 Hlavní cíle

Cílem diplomové práce je vybrání vhodných algoritmů řešících soustavy lineárních rovnic a nerovnic a tyto algoritmy implementovat. Dále rozebrat teoretické možnosti jejich paralelizace a tato paralelní řešení implementovat. Provést měření na různé škále možných vstupů a porovnat sekvenční řešení, paralelní řešení a řešení třetích stran. Nakonec zhodnotit paralelní řešení a doporučit, pro jaké vstupní systémy je dobré, je nasadit.

2.1.2 Funkční požadavky

- sekvenční algoritmy pro řešení soustav lineárních rovnic a nerovnic
- paralelní řešení těchto algoritmů
- měření času

2.1.3 Nefunkční požadavky

- použití programovacího jazyku C/C++
- využití knihovny OpenMP pro paralelizaci

$$A = \begin{pmatrix} L_1 & L_2 & \cdots & L_n \\ A_1 & & & \\ & A_2 & & \\ & & \ddots & \\ & & & A_n \end{pmatrix}$$

Obrázek 2.1: Block-angular struktura

2.2 Podobné práce a řešiče

2.2.1 Podobné práce

Na toto téma existuje spousta kvalitních článků a prací, proto zde zmíním jen ty, které se snaží o již zmíněnou paralelizaci.

2.2.1.1 Řešení soustav lineárních rovnic

Mezi práce, které pojednávají o paralelizaci řešení soustav lineárních rovnic, patří např. bakalářská práce [4]. Tato práce popisuje všechny možné metody pro řešení soustav lineárních rovnic, od těch nejzákladnějších jako je např. Gaussova eliminace (viz. 1.2.1.1) až po ty, které se v praxi používají, jako je např. Metoda sdružených gradientů (viz. 1.2.3.3). Autor se dále věnuje návrhu a implementaci paralelního počítání metody sdružených gradientů a vlastního nasazení na portál `webmath.zcu.cz`.

Další prací na toto téma je [18], kde autoři popisují způsob paralelizace Jacobiho iteračního algoritmu (viz. 1.2.2.1) pro řešení rozsáhlých CTMC systémů (jedná se o formalismus využívající Markovské řetězce a odpovídá řídkým soustavám lineárních rovnic).

2.2.1.2 Řešení soustav lineárních nerovnic

Článek [19] pojednává o paralelním řešení pro lineární programování pomocí Dantzig-Wolfe dekompozice (o této dekompozici si lze přečíst v [20]). Zjistilo se, že mnoho obrovských LP problémů má tzv. `block-angular` strukturu omezení (matice je rozdělena do nepřekrývajících se diagonálních submatic a pásem tvořeným neoddělitelnými částmi, viz. obr. 2.1). To umožňuje problém dekomponovat do dvouúrovňového paralelního řešení tak, že hlavní procesor rozdá jednotlivé diagonální submatice spolu s dalšími daty (vypočítanou hodnotu simplexového multiplikátoru na základě zbylého pásu, jež je vektor, kterým je potřeba přenásobit submatici před samotným výpočtem) k řešení ostatním procesorům a na základě výsledků od ostatních procesorů výpočet ukončí a vrátí optimum nebo započne další iteraci.

V roce 2007 vyšla zajímavá práce [21], která shrnuje všechny dosavadní pokusy o paralelizaci Simplexové metody (viz. 1.3.1.1). Práce začíná od klasické Simplexové metody a postupně se propracovává až k speciálním případům Revidovaných simplexových metod. Dále popisuje různé techniky paralelizace této metody. Autor v závěru konstatuje, že efektivní paralelizaci se podařilo provézt pouze na sekvenčních verzích klasické Simplexové metody a že pro praktické použití, kde převládají omezení tvořící řídkou matici, je nejlepším řešením stále sekvenční verze Revidované simplexové metody.

Práce [22] pojednává o paralelizaci Metoda větví a řezů (viz. 1.3.3.1), která se používá k řešení smíšených lineárních programů. Po rozebrání sekvenční verze algoritmu následuje detailní pojednání o možných přístupech k paralelizaci a samotné paralelizaci v podobě Symphony a Alps. Symphony a Alps jsou frameworky používané COIN-OR.

Dalšími zajímavými články zabývající se paralelizací LP jsou např. [23], jež popisuje pravděpodobnostní CRCW PRAM algoritmus s $O(n)$ procesory pro dosažení konstantní složitosti, nebo [24], kde hlavní myšlenkou algoritmu je opět dekompozice, ale snaží se o lepší škálovatelnost a za tímto účelem využívá hypergrafy.

2.2.2 Dostupné řešiče

Na internetu je k dispozici spousta řešičů, ale na hlavních stránkách se většinou vyskytují různé on-line nástroje, které umožňují řešit soustavy v řádu pouze desítek proměnných. Pro řešení soustav větších rozměrů lze sáhnout po komplexních matematických nástrojích nebo řešičích, které jsou přímo pro daný problém určeny. Pro řešení soustav lineárních rovnic lze použít např.:

- Wolfram Mathematica
- MATLAB
- Maple

a pro řešení soustav lineárních nerovnic lze použít např.:

- CBC od COIN-OR
- LP_Solve
- GLPK

2.2.3 Řešiče k porovnání

2.2.3.1 Wolfram Mathematica

Rozhodl jsem se porovnat implementovaný řešič pro řešení soustav lineárních rovnic s programem Wolfram Mathematica 10. Tento program jsem si vybral,

protože je hodně známý a velmi používaný. Mathematica umožňuje řešit soustavu lineárních rovnic pomocí funkce `LinearSolve`. Pokud se této funkci na vstup předloží pouze matice A a vektor b, funkce sama rozhodne, kterými metodami daný problém vyřeší. Funkce dále přijímá mimo jiné parametr `Method`, který se může nastavit na `Krylov` a dále na `ConjugateGradient`, jež odpovídá přímo metodě sdruženým gradientům (viz. 1.2.3.3).

Metoda `LinearSolve` je paralelizována pomocí vícevláknové knihovny MKL, ale z měření, které jsem prováděl, vyplývá, že přímo metoda `ConjugateGradient` paralelně neběží. Dále metoda umožňuje nastavit přesnost měření obalením matice a vektoru funkcí N s parametrem, který udává počet desetinných míst.

Matici je možné ukládat výčtem všech prvků v dvojrozměrném seznamu, jež odpovídá uložení matice jako plné (viz. 2.3.1.1). Dále ji je možné ukládat pomocí struktury `SparseArray`, kde jednotlivé nenulové prvky jsou dány jejich řádky a sloupci.

2.2.3.2 LP_Solve

Rozhodl jsem se porovnat implementovaný řešič pro řešení soustav lineárních nerovnic s programem lp_solve verze 5.5.2.0. Vybral jsem si ho, protože se jedná o známý řešič MILP problémů a je přímo pro tento problém dělaný. Program používá pro řešení Metodu větví a mezí (viz. 1.3.2.2) spolu se Simplexovou metodou (viz. 1.3.1.1). Bohužel lp_solve neumožňuje řešit problém paralelně, ale přesto věřím, že je natolik optimalizovaný, že bude těžké se mu vyrovnat. Tento program je vydáván pod licencí LGPL.

Program lp_solve kromě souboru s problémem může dostat na vstupu řadu parametrů, které přímo ovlivňují celkovou dobu výpočtu nebo jeho kvalitu. Mezi těmito parametry je např. parametr určující, jestli se jedná o minimalizaci nebo maximalizaci problému, parametr určující pivotáž (dle jakých pravidel se vybere klíčový sloupec a řádek v jedné iteraci simplexové metody), parametr nastavující škálování (ve smyslu normalizace vstupních hodnot pro urychlení výpočtu), parametr určující, která větev při větvení se vezme jako první, a spoustu dalších užitečných parametrů.

Soubory s problémy mohou být více formátů. Mezi tyto formáty patří např. `CPLEX LP`, `MPS` a `LP`.

2.3 Sekvenční řešení

2.3.1 Možnosti uložení dat

Matice i vektory bývají zpravidla velmi řídké při reálných inženýrských úlohách (počet nenulových prvků je velmi nízký), z tohoto důvodu se používají různé způsoby uložení dat, dále jen formáty. Protože na velikostech vektorů v porovnání s velikostmi matic téměř nezáleží, dále zmíněné formáty jsou formáty pro uložení matic. Těmito formáty jsou plná matice, kompresované

$$\begin{bmatrix} 1 & 0 & 5 & 4 \\ 0 & 4 & 0 & 0 \\ 5 & 0 & 0 & 2 \\ 4 & 0 & 2 & 3 \end{bmatrix}$$

Obrázek 2.2: Plná matice

řádky a symetrické kompresované řádky, které budou popsány níže. Mezi další známé formáty uložení matic patří např. skyline (proměnná šířka pásu) nebo pásová matice (formáty pro uložení matic převzaty z [25]).

2.3.1.1 Plná matice

Plná matice, jak už název napovídá, je uložení všech prvků matice včetně nulových. Toto uložení má smysl, pokud se jedná o hustou matici (není přesná definice, jedná se o matici, kde poměr nulových prvků se blíží nule) nebo pokud se očekává, že se matice bude často měnit. Ukázku plné matice lze vidět na obr. 2.2.

2.3.1.2 Kompresované řádky

Kompresované řádky je způsob uložení matice, kdy do prvního pole se zapíší všechny nenulové prvky matice (postupně řádek po řádku), do druhého pole se zapíší odpovídající sloupcové indexy těchto nenulových prvků a do třetího pole se zapíší adresy začátků řádků matice (kde daný řádek v prvním a ve druhém poli začíná). Toto uložení má smysl, pokud se jedná o řídkou matici (opak husté, poměr nenulových prvků se blíží nule), matice se takřka nemění (změna nenulového prvku na nulový a naopak si vyžádá posun prvků v poli a přeindexování) a očekává se práce s celým řádkem (přímá indexace si vyžádá $O(n)$ operací, neboť po nalezení správné části druhého pole, které obsahuje sloupcové indexy požadovaného řádku, je v nejhorším případě potřeba projít tuto část celou pro nalezení požadovaného sloupce). Proto jsou kompresované řádky vhodné zejména pro násobení matice vektorem (při násobení matice s vektorem se vždy bere celý řádek matice). Ukázku kompresovaných řádků lze vidět na obr. 2.3, odpovídající plnou matici lze vidět na obr. 2.2 (řádky a sloupce jsou indexovány od nuly).

2.3.1.3 Symetrické kompresované řádky

Symetrické kompresované řádky, jsou kompresované řádky pro symetrické matice. Pro symetrické matice si stačí pamatovat pouze horní nebo dolní trojúhelník matice (prvky pod nebo nad diagonálou a diagonála), dále se již bude

[1 5 4 4 5 2 4 2 3]
[0 2 3 1 0 3 0 2 3]
[0 3 4 6]

Obrázek 2.3: Kompresované řádky

[1 5 4 4 2 3]
[0 2 3 1 3 3]
[0 3 4 5]

Obrázek 2.4: Symetrické kompresované řádky

uvažovat pouze horní trojúhelník. Tento trojúhelník se pak ukládá stejně jako u kompresovaných řádků. Vlastnosti zůstávají stejné, jen je potřeba pro uložení pouze polovina paměti (není to celá polovina, neboť se ukládá i diagonála a třetí řádek s adresami je stále stejně dlouhý). Ukázku symetrických kompresovaných řádků lze vidět na obr. 2.4, odpovídající plnou matici lze vidět na obr. 2.2 (řádky a sloupce jsou indexovány od nuly).

2.3.2 Vybrané algoritmy

2.3.2.1 Řešení soustav lineárních rovnic

Pro řešení soustav lineárních rovnic jsem vybral Metodu sdružených gradientů (viz. 1.2.3.3). Tuto metodu jsem vybral na základě její vlastností a to, že průběžné řešení je známo po celou dobu běhu algoritmu (uživatel si bude moci před začátkem běhu algoritmu zvolit cílovou hodnotu ϵ) a že algoritmus skončí po nejvíce n krocích, kde n je velikost vstupní matice (uživatel má po konečném počtu kroků k dispozici přesné řešení).

Dále jsem si jí vybral hlavně proto, že má dle mého názoru z metod pro řešení soustav lineárních rovnic (viz. 1.2) nejblíže k řezům. Řezy se zde objeví na úrovni násobení matice s vektorem, kde jednotlivé skupiny řádků mohou být řešeny paralelně. Nejsou to řezy v pravém slova smyslu, ale řezy v pravém slova smyslu se ani u řešení soustav lineárních rovnic neobjevují.

Nevýhodou této metody je, že na vstupu je potřeba symetrická positivně definitní matice. Přestože se toto zdá jako veliké omezení, veliké množství reálných inženýrských úloh vede na soustavu s takovouto maticí (úlohy z termodynamiky, mechaniky a dalších).

Algoritmus je poměrně jednoduchý (viz. alg. 1.3), proto jsem se rozhodl v této práci zabývat různými formáty uložení vstupní matice a jaké mají tyto

formáty vliv na paměť a rychlost algoritmu (včetně paralelního řešení). Těmito formáty jsou plná matice, kompresované řádky a symetrické kompresované řádky (viz. 2.3.1).

Vektor se ukládá jako plný (všechny prvky včetně nenulových), neboť jiný formát by akorát způsobil nárůst paměti (byly by potřeba dvě pole, kde jedno by obsahovalo nenulové prvky a druhé sloupcové, respektive řádkové indexy a pokud by vektor měl více jak polovinu prvků nenulových, bylo by potřeba již více paměti než v plném uložení, což při operacích, které algoritmus používá, je skoro pokaždé).

2.3.2.2 Řešení soustav lineárních nerovnic

Pro soustavy lineárních nerovnic jsem vybral Metodu větví a mezí (viz. 1.3.2.2). Tuto metodu jsem vybral, protože umí řešit i MILP (viz. 1.3.3), takže má uživatel možnost si určit, které proměnné jsou celočíselné a které jsou reálné. Na rozdíl od metod sečných nadrovin (viz. 1.3.2.3), se tato metoda v každé iteraci větví a jednotlivé větve již mohou být paralelně zpracovány.

Další výhodou této metody je ta, že průběžně nalézá suboptimální řešení, které je možné vrátit uživateli. Díky této vlastnosti a generované stromové struktuře je možné přidat parametry omezující maximální hloubku zanoření, maximální počet suboptimálních řešení, minimální a maximální cenu řešení nebo např. ukončit výpočet a vrátit nejlepší nalezené řešení po uplynutí určité doby.

Pro řešení LP v rámci první iterace se použije standardně Simplexový algoritmus (viz. alg. 1.4) a následně v dalších iterací po přidání nových omezení se použije jeho duální verze (viz. alg. 1.5), jež jak bylo zmíněno, umožňuje navázat na předchozí výpočet. Protože oba tyto algoritmy pracují nad simplexovou tabulkou (viz. obr. 1.7), kde je potřeba přičítat násobky řádků k jiným řádkům, je vhodné uložení této tabulky v plném formátu (viz. 2.3.1.1).

Metoda větví a mezí, na rozdíl od metody sdružených gradientů, kterou jsem vybral pro řešení soustav lineárních rovnic, je úlohou řezů v pravém slova smyslu. S každým novým přidaným omezením (iterací) se aktuální prostor možných řešení rozdělí na dvě části, které jsou dále rekurzivně děleny.

2.4 Paralelní řešení

2.4.1 Paralelní architektura

Pro úvahy a počítání v dalších podsekcích je potřeba zadefinovat paralelní architekturu, respektive abstraktní model počítače, na kterém výsledné algoritmy poběží. Tímto modelem je PRAM, konkrétně PRAM CREW. Následující popis tohoto modelu je převzat z [26].

Definice 2.4.1 *PRAM model je model, kde každá operace trvá jednotkový čas (všechny operace trvají stejně dlouho a jsou vykonány v konstantním čase bez*

ohledu na délku operandů), má neomezený počet procesorů p a sdílených paměťových buněk m a každý procesor má vlastní lokální paměť. Vstupem algoritmu je n položek v prvních n buňkách sdílené paměti a výstup je n' položek v n' buňkách sdílené paměti.

Tento model dále rozeznává 3 typy operací: čtení buňky sdílené paměti, lokální operace a zápis buňky sdílené paměti. Tento jednoduchý a intuitivní model slouží především k otestování, zda má smysl algoritmus vůbec paralelizovat, neboť pokud nebude rozumně paralelizován na tomto modelu, tak nebude ani na jakékoliv jiné paralelní architektuře.

Dále se PRAM model dělí na PRAM EREW, PRAM CREW a PRAM CRCW. Model PRAM EREW zakazuje čtení i zápis téže sdílené paměťové buňky dvoum a více procesorům v jeden okamžik. Model PRAM CREW umožňuje paralelní čtení sdílené paměťové buňky, ale stále zakazuje zápis do téže buňky více procesorům a model PRAM CRCW dovoluje paralelní čtení i zápis (dále se dělí podle způsobu zapsání do téže paměťové buňky na prioritní, náhodný a shodný).

2.4.2 Možnosti paralelizace

V této sekci jsou rozebrány jednotlivé možnosti paralelizace pro vybrané algoritmy (viz. 2.3.2). Pro další použití níže je zde uvedena definice paralelní redukce a stavového prostoru. Dále je zde popsán Dijkstrův algoritmus pro distribuované ukončení paralelního výpočtu [27].

Definice 2.4.2 *Paralelní redukce je operace, kdy je potřeba n vstupních hodnot redukovat na jedinou hodnotu pomocí více procesorů p. Jedná se o výpočet $a = a_1 \oplus a_2 \oplus \ldots \oplus a_n$, kde operace $\oplus$ je asociativní (nezáleží na pořadí výpočtu). Složitost této operace je $O(\frac{n}{p} + \log p)$.*

Definice 2.4.3 *Stavový prostor algoritmu A řešícího problém P je dvojice (S, Q), kde S je množina všech možných stavů s_i algoritmů A, která řeší problém P, a Q je množina operací q_j typu $S \to S$, kde $q_j(s_i) \neq s_i$ pro všechna q_j a s_i.*

Dijkstrův algoritmus pro distribuované ukončení paralelního výpočtu pracuje tak, že každý procesor může být obarven bílou nebo černou barvou a dále je mezi procesory posílán pešek (v originále `token`), který opět může být obarven bílou nebo černou barvou. Všechny procesory jsou na počátku obarveny bílou barvou. Jakmile procesoru P_0 dojde práce, posílá procesoru P_1 peška obarveného na bílo a sám se obarví na bílo. Pokud procesor P_i pošle práci procesoru P_j, kde $j < i$, obarví se na černo. Pokud procesor P_i dostane peška a je černý, obarví peška na černo. Jakmile procesoru P_i dojde práce, posílá procesoru P_{i+1} peška a obarví se na bílo (poslední procesor posílá prvnímu procesoru). Pokud procesor P_0 obdrží bílého peška, práce je hotova a všechny

$$
\begin{array}{c|c|}
P_1 & \\ \hline
P_2 & \\ \hline
\dots & \\ \hline
P_p & \\
\end{array}
\cdot
\begin{bmatrix} \\ \\ \\ \\ \end{bmatrix}
=
\begin{bmatrix} f(P_1) \\ \hline f(P_2) \\ \hline f(\dots) \\ \hline f(P_p) \end{bmatrix}
$$

Obrázek 2.5: Paralelizace násobení matice s vektorem po řádcích

procesory mohou skončit, pokud obdrží černého peška, obarví ho na bílo a započne nové kolo.

2.4.2.1 Řešení soustav lineárních rovnic

Vybraná metoda sdružených gradientů (viz. 1.2.3.3) spočívá převážně v násobení matice vektorem a ve vektorových operacích (viz. alg. 1.3). Tyto operace se poměrně dobře paralelizují, neboť výpočty jednotlivých složek výsledného vektoru těmito operacemi jsou na sobě nezávislé. Výjimkou je skalární součin dvou vektorů, kde je zapotřebí provést paralelní redukci.

Při násobení matice vektorem i-tá složka výsledného vektoru vznikne skalárním součinem i-tého řádku násobené matice a násobeným vektorem (zde je skalární součin počítán pouze jedním procesorem, proto není potřeba žádné paralelní redukce), proto má smysl paralelizovat po řádcích (při paralelizaci po sloupcích by i-tá složka výsledného vektoru vznikla sumou mezivýsledků pro daný řádek i od všech procesorů, jež by vedlo na paralelní redukci a na zbytečnou režii navíc). Schéma paralelizace násobení matice s vektorem po řádcích lze vidět na obr. 2.5 a po sloupcích na obr. 2.6 (funkce f reprezentuje výsledek pro daný blok a jejími parametry jsou procesory, které jsou pro daný blok relevantní neboli chtějí do daného bloku zapisovat).

Dále je třeba vzít v úvahu konkrétní způsob uložení matice, jež bylo zmíněno v předchozí sekci, a jaký to má vliv na samotné násobení matice s vektorem (viz. níže).

Nakonec je potřeba zmínit, že jednotlivé paralelizace mají smysl až od určité velikosti vstupní instance. Pokud je instance na vstupu příliš malá, nemá smysl vůbec paralelizovat, neboť by režie na rozdělení práce a komunikaci byla větší než samotný sekvenční běh. Pro větší instanci již může mít smysl paralelizovat násobení matice s vektorem, ale pro vektorové operace může být instance stále natolik malá, že samotný běh algoritmu brzdí. Až od určité velikosti instance má smysl paralelizovat na všech úrovní, proto je třeba mít

$$\begin{array}{c|c|c|c} P_1 & P_2 & \dots & P_p \end{array} \cdot \left[\ \right] = \begin{bmatrix} f(P_1, P_2, \dots, P_p) \\ \hline f(P_1, P_2, \dots, P_p) \\ \hline f(P_1, P_2, \dots, P_p) \\ \hline f(P_1, P_2, \dots, P_p) \end{bmatrix}$$

Obrázek 2.6: Paralelizace násobení matice s vektorem po sloupcích

možnost neparalelizovat vůbec, paralelizovat pouze násobení matice s vektorem a paralelizovat jak násobení matice s vektorem, tak i jednotlivé vektorové operace.

Plná matice Při uložení matice jako plné (viz. 2.3.1.1) dostane každý procesor přidělen statický počet řádků $\frac{n}{p}$, kde n je velikost matice (jedná se o čtvercovou matici, která má stejný počet řádků jako sloupců) a p je počet procesorů. V tomto případě není dále co řešit, neboť jednotlivé výpočty trvají stejně dlouho (pomineme-li, že několik x posledních procesorů dostane o řádek méně, neboť velikost matice n nemusí být dělitelná počtem procesorů p). Jiný způsob rozdělování práce (např. dynamicky - po vypočtení r řádků zažádat o další řádky) zde nemá smysl (výpočty trvají stejně dlouho), neboť by docházelo akorát ke zbytečné komunikační režii navíc.

Kompresované řádky Při uložení matice pomocí kompresovaných řádků (viz. 2.3.1.2) je již potřeba vzít v úvahu, že ne všechny řádky matice mají stejný počet nenulových hodnot (pokud budou nenulové prvky v matici přibližně rovnoměrně rozděleny, není třeba se tímto problémem dále zabývat) a tudíž se mohou doby výpočtu jednotlivých procesorů značně lišit (při násobení s kompresovanými řádky se prochází pouze nenulové hodnoty a v extrémním případu by mohl jeden či více procesorů dostat úplně prázdné řádky a tudíž by během celého výpočtu nic nedělaly a čekaly až ostatní procesory dopočítají, což je nežádoucí). Proto je potřeba již zvolit dynamické přidělování řádků a to tak, aby práce mezi procesory byla rozvržena co možno nejrovnoměrněji, ale aby komunikační režie nevzrostla natolik, že by celkový výpočet trval ještě déle než při statickém rozdělení (to znamená, že je potřeba zvolit počet přidělených řádků v jedné iteraci r, kde $r \in (1, \frac{n}{p})$ a platí, že pro $r = 1$ je práce rozdělena nejrovnoměrněji, ale je potřeba stálá komunikace při rozdělování, a

pro $r = \frac{n}{p}$ se jedná o statické rozdělení, kde není potřeba žádné komunikace, ale práce je rozdělena nejnerovnoměrněji).

Symetrické kompresované řádky Pro uložení matice pomocí symetrických kompresovaných řádků (viz. 2.3.1.3) platí to samé co pro kompresované řádky (zde je to možná více znatelné, neboť procesor, který dostane první řádky, může mít řádky dlouhé až n a procesor, který dostane poslední řádky, může mít řádky, které budou dlouhé pouze jednotky). Navíc je potřeba vyřešit problém spojený s tím, že i-tý řádek takto uložený obsahuje hodnoty pouze od diagonály do konce řádku a tento řádek současně reprezentuje i hodnoty v i-tém sloupci (to znamená, že procesor už nemá složku výsledného vektoru, do které zapisuje výsledek součinu přiděleného řádku s násobeným vektorem, exkluzivně pro sebe). Tento problém se dá vyřešit tak, že jednotlivé složky výsledného vektoru budou v kritické sekci (tento způsob by byl vhodný pro opravdu řídké matice, kde zápis více procesorů do jedné a téže složky by byl velmi zřídka) nebo pro každý procesor bude existovat vlastní výsledný vektor, do kterého bude zapisovat, a na konci se odpovídající hodnoty na stejných řádcích paralelně sečtou do jednoho výsledného vektoru (každý procesor si pak může vzít staticky přidělenou skupinu řádků, kterou sečte do výsledného vektoru na odpovídající místa).

2.4.2.2 Řešení soustav lineárních nerovnic

Vybraná metoda větví a mezí (viz. 1.3.2.2) postupně generuje stromovou strukturu (binární strom, neboť každý uzel má nanejvýš 2 potomky), jež odpovídá úloze prohledávání stavového prostoru (v dalších odstavcích se již převážně bude používat tento pojem). V rámci jednoho uzlu (stavu) se nad aktuální simplexovou tabulkou provede výpočet a pokud se nenalezne řešení (nepřípustný prostor nebo neomezené řešení) nebo je optimum tohoto řešení menší než dosavadní nalezené celočíselné optimum, větev končí, jinak se dále větví, jak je zmíněno v popisu samotné metody.

Smysl má paralelizovat pouze prohledávání stavového prostoru, protože počet stavů ku simplexové tabulce roste exponenciálně (při větvení v dané hloubce do tabulky přibude pouze jeden řádek, ale počet stavů se zdvojnásobí) a proto je tabulka ku velikosti prostoru zanedbatelná a dále protože je lepší paralelizovat ve vyšších úrovních, aby nedocházelo tak často ke komunikační režii (kdyby se paralelizovalo v rámci simplexového algoritmu, respektive duálního a komunikační režie v rámci jednoho uzlu by stála pouze 1 operaci, tak by mohlo dojít až ke $2^{k+1} - 1$ komunikačních operacích, což odpovídá maximálnímu počtu uzlů binárního stromu v hloubce k).

Při paralelizaci prohledávání stavového prostoru, kde úkolem je najití optima úlohy, je dále potřeba vyřešit řadu otázek. Těmito otázkami jsou počáteční dělba práce, přerozdělování práce v průběhu algoritmu (vyvažování zátěže), sdílení globálního optima a ukončení výpočtu (ukončení výpočtu se

provede Dijkstrovým algoritmem popsaným na začátku této sekce). V obecném případě je potřeba ještě řešit hloubku prohledávaného prostoru (problém nemusí být na hloubku omezený nebo mohou vznikat cykly), což u úlohy MILP není potřeba řešit (nejhorší možný případ je řešit ILP, kdy je potřeba najít všechny proměnné celočíselné, ale i v tomto případě algoritmus skončí po konečném počtu kroků, neboť se při každém větvení snižuje počet možných řešení do dalších větvích a těchto možných řešení je na počátku omezeně mnoho).

Počáteční dělba práce V obecném případu probíhá počáteční dělba práce tak, že hlavní procesor do svého zásobníku přidá počáteční stav a provede expanze do takové hloubky (může expandovat i do šířky), že na zásobníku bude mít dostatek práce (stavů), aby je mohl rozdělit mezi všechny procesory. Při řešení úlohy MILP to znamená, že procesor P_0 vyřeší počáteční simplexovou tabulku pomocí simplexového algoritmu a dále po rozvětvení si jednu větev nechá a druhou větev si uloží na zásobník, respektive její simplexovou tabulku. Ponechanou větev dále řeší pomocí duálního algoritmu a takto pokračuje, dokud zásobník nemá dostatečně veliký, aby mohl dát práci všem procesorům.

Přerozdělování práce Přerozdělování práce v průběhu algoritmu je asi nejcitlivějším místem paralelizace. Všechny procesory by měly být pokud možno neustále vytíženy (konat nějakou užitečnou práci) a přitom komunikační režie by měla být co možná nejmenší. Tohoto cíle je velmi těžké dosáhnout, neboť zpravidla není předem známá výsledná podoba prohledávaného stromu a může se snadno stát, že část přerozdělené práce je vykonána ihned a procesory mezi sebou většinu času pouze komunikují a přerozdělují si práci.

Když procesoru dojde práce, má více možností, jak požádat o novou. Může požádat o práci všechny ostatní procesory, vybranou množinu procesorů nebo jen jeden procesor. Při výběru množiny procesorů nebo jen jedno procesoru pak může volit zcela náhodně, cyklicky (ptá se vždy procesoru s vyšším indexem než se ptal naposled a po posledním procesoru se ptá zase prvního procesoru) nebo podle aktuální velikosti zásobníku (pokud je možné ho zjistit bez nutnosti další komunikace). Po získání práce se může ihned opět pustit do výpočtů.

Dále se všechny procesory jednou za k iterací podívají na příchozí požadavky o práci a podle počtu požadavků rozdělí svůj zásobník a předají práci (zvolené k má přímý vliv na vyvažování zátěže a množství komunikační režie). Z příchozích požadavků mohou vybrat jen ty, které v aktuální moment nemají žádnou práci (pokud je to opět možné zjistit bez nutnosti další komunikace), a dále vybrat jen podmnožinu procesorů, kterým práci předá (při nedostatku práce může být lepší vyhovět jen části požadavkům s tím, že si procesor ponechá rezervu, aby vzápětí sám nežádal o práci). Dále je možné dělit zásobník mezi vybrané procesory rovnoměrně nebo nerovnoměrně (např. procesor si

sám nechá $\frac{1}{2}$ práce, prvnímu žadateli pošle $\frac{1}{4}$ práce, druhému $\frac{1}{8}$ práce, atd.). Pokud je zásobník tříděn podle hloubky zanoření jednotlivých stavů, je možné vzít ještě v úvahu, že stavy s nižším hloubkou (málo rozexpandované) obsahují více práce něž stavy, které jsou hodně hluboko, a dělit zásobník tak, že množství práce bude stanoveno za základě hloubky stavů a ne podle počtu stavů.

Sdílení globálního optima Sdílení globálního optima se dá v podstatě dělat pouze dvěma způsoby, buď se nedělá a každý procesor si drží vlastní lokální optimum a po skočení všech procesorů se paralelní redukcí nalezne globální nebo se dělá a drží se globální optimum, které je pokaždé aktualizováno při najití nového globálního optima a každý procesor po k iteracích si své lokální optimum aktualizuje na hodnotu globálního optima. Výhoda prvního přístupu je, že nedochází k tak časté komunikační režii, a nevýhodou je, že stavový prostor se méně ořezává, u druhého přístupu je tomu právě naopak.

2.4.3 Výsledek

V této sekci dále se počítá paralelní zrychlení pro navržené algoritmy na paralelní architektuře PRAM CREW (viz. 2.4.1). Protože se jedná o abstraktní model, výpočty níže jsou čistě teoretické a slouží spíše jako indície při rozhodování, zda má úlohu smysl vůbec paralelizovat. Pro výpočet paralelního zrychlení je u obou problému nejdříve potřeba vypočítat sekvenční čas a poté paralelní čas.

2.4.3.1 Řešení soustav lineárních rovnic

Pro výpočet sekvenčního času $T(n)$ a paralelního času $T(n, p)$ u Metody sdružených gradientů (viz. 1.2.3.3) pro jednotlivé způsoby uložení matice, je potřeba se podívat do jejího pseudokódu (viz. alg. 1.3). Počáteční rozdíl vektoru b a přenásobené matice A vektorem x_0 ($b - Ax_0$), není potřeba započítávat, protože počáteční vektor x_0 se pokládá roven nulovému vektoru a tudíž výsledkem těchto operací bude vektor b (i kdyby se počáteční vektor x_0 nastavoval různý od nulového vektoru, pořád by tyto operace byly zanedbatelné v porovnání se smyčkou, která následuje). Parametr ϵ se nastaví roven nule, protože se počítá s nejdéle trvajícím možným výsledkem (pro připomenutí $\epsilon = 0$ odpovídá přesnému řešení). Dále je možné si pamatovat jednotlivé mezivýsledky, aby zbytečně nedocházelo k redundantním výpočtům. Upravený pseudokód je zachycen v alg. 2.1.

Dále je potřeba zavést parametry k, ρ_m, ρ_r a r. Parametr k reprezentuje komunikační režii pro přidělení jednoho bloku práce jednomu procesoru v paralelní verzi algoritmu (tuto práci vykonává hlavní procesor a sám se do tohoto počtu nepočítá, pouze ostatní procesory). Parametr ρ_m reprezentuje hustotu nenulových prvků matice A a platí $\rho_m \in (0, 1)$ (pro $\rho_m = 1$ matice A

Algoritmus 2.1 Metoda sdružených gradientů - upravená verze

function METODASDRUZENYCHGRADIENTU(A, b)
  $d = r = b$
  $x = 0$
  $[rr] = r^T r$

  while $[rr] > 0$ **do**
    $[Ad] = Ad$
    $\alpha = \frac{[rr]}{d^T[Ad]}$
    $x = x + \alpha d$
    $r = r - \alpha[Ad]$
    $[RR] = [rr]$
    $[rr] = r^T r$
    $\beta = \frac{[rr]}{[RR]}$
    $d = r + \beta d$

  return x

obsahuje pouze nenulové prvky a pro $\rho_m = 0$ pouze nulové). Parametr ρ_r je roven maximální hustotě nenulových prvků v řádku a parametr r byl definován v předchozí sekci (parametr udávající, kolik procesory dostanou přiděleno řádků matice v jedné iteraci v paralelní verzi algoritmu, kde $r \in (1, \frac{n}{p})$).

Pro jednodušší výpočet se dále bude počítat pouze s lokálními operacemi, respektive nebudou se započítávat operace čtení a zápisu ze sdílených buněk. Toto je možné z toho důvodu, že matici stačí do lokální paměti načíst pouze jednou (matice se po dobu výpočtu nemění) a načítání a ukládání dat u vektorů pouze zvyšuje výslednou konstantu u operací, která nemá zásadní vliv na celkový výpočet.

Plná matice Pro uložení matice jako plné (viz. 2.3.1.1) platí, že parametry ρ_m a ρ_r nehrají roli, proto se s nimi nebude počítat (algoritmus pro tento typ uložení nerozeznává nulové a nenulové prvky). Parametr r také nehraje roli při uložení matice jako plné, neboť se používá statické přidělení a žádné iterace na přidělování dalších řádků zde neprobíhají.

Pro sekvenční čas $T(n)$ je potřeba vypočítat počet operacích provedených v rámci smyčky. V nejhorším možném případu smyčka skončí po n-ti iteracích (více v popisu metody). Dále je uvnitř smyčky potřeba 1x násobení matice s vektorem (Ad), 1x skalární součin vektorů ($r^T r$), 2x dělení reálných čísel (α a β) a 7x vektorová operace ($d^T[Ad]$, αd, $x + \alpha d$, $\alpha[Ad]$, $r - \alpha[Ad]$, βd a $r + \beta d$).

Skalární součin dvou vektorů si vyžádá $2n-1$ operací (je potřeba provést n násobení mezi jednotlivými složkami vektorů a následně tyto součiny pomocí

$n-1$ operací sečíst), násobení matice s vektorem $n(2n-1)$ operací (jedná se skalární součin vektorů pro n řádků) a jedna vektorová operace n (je potřeba provést n operací mezi odpovídajícími si složkami vektorů, respektive vektoru a skaláru). Pro celkový čas sekvenčního algoritmu tedy platí

$$\begin{aligned} T(n) =& n(n(2n-1)+2n-1+7n+2) \\ =& 2n^3+8n^2+n. \end{aligned} \tag{2.1}$$

Platí $T(n)=O(n^3)$, jež je asymptotická mez Gaussovy eliminace (viz. 1.2.1.1).

Pro paralelní čas $T(n,p)$ je potřeba vyjádřit, jak dlouho budou trvat výše jmenované sekvenční operace pro p procesorů. Skalární součin dvou vektorů si nyní vyžádá $2\frac{n}{p}-1+\log(p)+k(p-1)$ operací (výraz $2\frac{n}{p}-1+\log(p)$ odpovídá paralelní redukci, $2\frac{n}{p}-1$ je sekvenční část prováděna každým procesorem lokálně a $\log(p)$ je předání a sečtení mezivýsledků do celkového výsledku, $k(p-1)$ odpovídá komunikační režii na rozdělení řádků do p bloků). Na násobení matice s vektorem je nyní potřeba $\frac{n(2n-1)}{p}+k(p-1)$ operací (podobně jako u paralelní redukce výraz $\frac{n(2n-1)}{p}$ odpovídá sekvenční části prováděné paralelně p procesory, ale zde není potřeba výsledek redukovat, proto se ve výrazu neobjevuje žádné $\log(p)$). Nakonec na jednu vektorovou operaci je nyní potřeba $\frac{n}{p}+k(p-1)$ operací. Pro celkový čas paralelní verze platí

$$\begin{aligned} T(n,p) =& n(\frac{n(2n-1)}{p}+k(p-1)+2\frac{n}{p}-1+\log(p)+k(p-1) \\ & +7(\frac{n}{p}+k(p-1))+2) \\ =& \frac{1}{p}(2n^3+8n^2)+9nk(p-1)+n\log(p)+n, \end{aligned} \tag{2.2}$$

kde výraz $\frac{1}{p}(2n^3+8n^2)$ odpovídá sekvenční části, která lze zparalelizovat, n odpovídá sekvenční části, která nelze zparalelizovat, a výraz $9nk(p-1)+n\log(p)$ odpovídá komunikační režii.

Pro paralelní zrychlení dané vztahem $S(n,p)=\frac{T(n)}{T(n,p)}$ poté platí

$$\begin{aligned} S(n,p) =& \frac{2n^3+8n^2+n}{\frac{1}{p}(2n^3+8n^2)+9nk(p-1)+n\log(p)+n} \\ =& \frac{2n^2+8n+1}{\frac{1}{p}(2n^2+8n)+9k(p-1)+\log(p)+1}. \end{aligned} \tag{2.3}$$

Z úpravy lze vidět, že zrychlení se provádí v rámci jedné iterace a počet iterací nemá vliv na celkové zrychlení. Kontrolou výpočtů může být dosazení $p=1$ do vzorců pro paralelní čas $T(n,p)$ a paralelní zrychlení $S(n,p)$, výsledkem je $T(n,1)=T(n)$ a $S(n,1)=1$, což je očekávaný výsledek.

Kompresované řádky Pro uložení matice pomocí kompresovaných řádků (viz. 2.3.1.2) se předchozí výrazy pro sekvenční čas $T(n)$, paralelní čas $T(n,p)$ a paralelní zrychlení $S(n,p)$ změní pouze v části odpovídající násobení matice s vektorem, zbytek zůstane stejný. Změna spočívá v tom, že je potřeba vzít v úvahu ještě parametry ρ_m, ρ_r a r.

Na násobení matice s vektorem pro kompresované řádky pro sekvenční čas $T(n)$ je potřeba $n(\rho_m 2n - \lceil \rho_m \rceil)$ operací (pro n řádků je potřeba provést skalární součin, u kterého stačí brát pouze nenulové prvky, a výraz $\lceil \rho_m \rceil$ je zde proto, že pokud bude ρ_m rovno nule nebude potřeba žádná operace a pokud nebude, bude vždy potřeba o operaci méně, neboť na součet n čísel je potřeba pouze $n-1$ sčítání).

Pro paralelní čas $T(n,p)$ tohoto násobení je potřeba $\frac{n(\rho_m 2n - \lceil \rho_m \rceil)}{p} + (\rho_r - \rho_m)nr\lceil\frac{p-1}{p}\rceil + k(p-1)\frac{n}{pr}$ operací. Výraz $\frac{n(\rho_m 2n - \lceil \rho_m \rceil)}{p}$ odpovídá rovnoměrnému rozdělení práce mezi všechny procesory (každý procesor dostane bloky, jejichž celková průměrná hustota je rovna ρ_m). Výraz $(\rho_r - \rho_m)nr\lceil\frac{p-1}{p}\rceil$ reprezentuje počet operací navíc, pokud nenulové prvky v přidělených blocích nebudou rovnoměrně rozděleny, respektive některý procesor bude mít více počítání. Část $\lceil\frac{p-1}{p}\rceil$ je tu proto, protože pokud je procesor pouze jeden, nemusí čekat na ostatní procesory, než dokončí svou práci, respektive celý výraz nabývá hodnoty 0, v opačném případu nabývá hodnoty 1. Část $(\rho_r - \rho_m)n$ říká, kolik operací navíc je potřeba pro řádek s maximální hustotou a r je tu proto, neboť v nejhorším možném případu dostane jeden procesor všech r řádků v daném bloku s maximální hustotou. Výraz $k(p-1)\frac{n}{pr}$ odpovídá komunikaci při statickém rozdělení s tím rozdílem, že je navíc přenásobený koeficientem $\frac{n}{pr}$, který udává, kolikrát víc tato komunikace proběhne v důsledku dynamického rozdělování. Pro kompresované řádky tedy platí

$$
\begin{aligned}
T(n) =& \rho_m 2n^3 + (9 - \lceil \rho_m \rceil)n^2 + n, \\
T(n,p) =& \frac{1}{p}(\rho_m 2n^3 + (9 - \lceil \rho_m \rceil)n^2) + (\rho_r - \rho_m)n^2 r\lceil\frac{p-1}{p}\rceil \\
& + (8 + \frac{n}{pr})nk(p-1) + n\log(p) + n, \\
S(n,p) =& \frac{\rho_m 2n^2 + (9 - \lceil \rho_m \rceil)n + 1}{\left(\begin{aligned} \tfrac{1}{p}(\rho_m 2n^2 + (9 - \lceil \rho_m \rceil)n) + (\rho_r - \rho_m)nr\lceil\tfrac{p-1}{p}\rceil \\ + (8 + \tfrac{n}{pr})k(p-1) + \log(p) + 1 \end{aligned}\right)}.
\end{aligned} \tag{2.4}
$$

Opět platí $T(n,1) = T(n)$ a $S(n,1) = 1$ a navíc při dosazení $\rho_m = 1$ a $r = \frac{n}{p}$ budou vzorce odpovídat vzorcům pro matici uloženou v plném formátu (pokud bude $\rho_m = 1$, tak i $\rho_n = 1$ a výraz $\rho_r - \rho_m$ bude roven nule).

Symetrické kompresované řádky Pro symetrické kompresované řádky bude sekvenční čas odpovídat sekvenčnímu času kompresovaným řádkům. U paralelního času, pokud se při násobení matice vektorem nepoužije kritická

sekce, ale vektory pro zapisování mezivýsledků, je potřeba ještě připočítat režii na redukci mezivýsledků neboli je potřeba paralelně sečíst p vektorů na odpovídajících si řádcích, kterých je n. To si vyžádá $\frac{n(p-1)}{p}$ operací navíc. Pro symetrické kompresované řádky tedy platí

$$\begin{aligned} T(n) =&\rho_m 2n^3 + (9 - \lceil \rho_m \rceil)n^2 + n, \\ T(n,p) =&\frac{1}{p}(\rho_m 2n^3 + (9 - \lceil \rho_m \rceil)n^2) + (\rho_r - \rho_m)n^2 r \lceil \frac{p-1}{p} \rceil \\ &+ (8 + \frac{n}{pr})nk(p-1) + n\log(p) + n + \frac{1}{p}n^2(p-1), \\ S(n,p) =&\frac{\rho_m 2n^2 + (9 - \lceil \rho_m \rceil)n + 1}{\left(\begin{array}{l} \frac{1}{p}(\rho_m 2n^2 + (9 - \lceil \rho_m \rceil)n) + (\rho_r - \rho_m)nr\lceil \frac{p-1}{p} \rceil \\ \quad + (8 + \frac{n}{pr})k(p-1) + \log(p) + 1 + \frac{1}{p}n(p-1) \end{array} \right)} \end{aligned} \tag{2.5}$$

a opět platí $T(n,1) = T(n)$ a $S(n,1) = 1$.

Je důležité si uvědomit, že i když se ze vzorců zdá, že symetrické kompresované řádky budou pomalejší, do vzorců není započítáno načítání a ukládání dat, jež bude u symetrických kompresovaných řádků téměř polovina.

2.4.3.2 Řešení soustav lineárních nerovnic

Oproti předchozím výpočtům pro metodu sdružených gradientů je nutné si uvědomit, že metoda větví a mezí, jak již bylo zmíněno v předešlé sekci, vede na prohledání stavového prostoru, který nemá jasně definovanou topologii, a tudíž podobné výpočty se provádějí velmi těžko.

První problém spočívá v tom, že při větvení není známo, v jakém poměru se stavový prostor rozdělí (navíc tento poměr se v každém větvení může lišit). Další problém je, že strom je průběžně ořezáván, jež napomáhá takřka nahodilé topologii. Dále úlohou může být najít řešení s reálnými hodnotami (LP) nebo druhý extrém řešení celočíselné (ILP) nebo někde mezi (MILP), jež má přímý vliv na dobu výpočtu. Díky zmíněným problémům může docházet dokonce k superlineárnímu zrychlení (viz. 1.4.1). A toto jsou jenom problémy spojené se sekvenčním řešení.

Z těchto důvodů zde nebudou uvedeny výpočty pro paralelní zrychlení, neboť by byly nicneříkající (již u metody sdružených gradientů pro kompresované řádky a symetrické kompresované řádky jsou dané výpočty spíše odhady). Pro více informací ohledně asymptotické časové složitosti pro sekvenční verze je tu článek [28] a článek [29] zabývající se Metodou větví a řezů pro problém obchodního cestujícího.

Kapitola 3

Realizace

Tato kapitola se zaobírá implementací algoritmů popsaných v předešlé kapitole pro programovací jazyk C++ s využitím knihovny OpenMP pro paralelizaci. Programovací jazyk C++ byl vybrán po svůj vysoký výkon v oblasti matematický výpočtů a knihovna OpenMP pro svoji jednoduchost a za účelem otestování, zda tato kombinace se může vyrovnat známým komerčním řešičům. Práce počítá se základními znalostmi tohoto jazyka a knihovny, a proto zde nebude popsán kompletní zdrojový kód, ale pouze nejdůležitější části, struktury a myšlenky.

3.1 Knihovna OpenMP

Jak již bylo zmíněno, cílem práce je předvést možnou paralelizaci pomocí knihovny OpenMP. OpenMP je knihovna pro víceprocesorové (vícevláknové) programování na sdílené paměti pro programovací jazyky C, C++ a Fortran. Výhodou této knihovny je, že pro paralelizaci sekvenčního kódu je zapotřebí pouze přidat speciální direktivy (např. `#pragma omp parallel`) na místa, která se mají provádět paralelně. Díky této vlastnosti se sekvenční kód téměř vůbec nezmění a je opravdu snadné implementovat paralelizaci. Nevýhodou je, že toto snadné použití nelze ve všech případech použít (bude popsáno v sekci pro řešení soustav lineárních rovnic).

Všechny direktivy jsou uvozeny pomocí `#pragma omp` a dále mohou obsahovat klauzule, které mají různý význam. Dále jsou popsány direktivy a klauzule, které jsou použity pro paralelní implementaci (viz. 3.3). Další nebo detailnější informace jsou dostupné z originálních stránek této knihovny `openmp.org`.

3.1.1 Direktivy

3.1.1.1 Parallel

Tato direktiva říká, že následující blok kódu se bude provádět paralelně. Vytvoří se tým, n-tice vláken, které budou provádět následující kód paralelně, kde n se obvykle bere z počtu jader. Každé vlákno bude provádět stejný kód a bude mít vlastní lokální proměnné uvnitř bloku. Defaultně jsou všechny proměnné vně bloku sdílené (programátor musí počítat s tím, že do nich mohou zapisovat všechna vlákna).

3.1.1.2 For

Tato direktiva říká, že následující `for`-cyklus se rozdělí mezi aktuální tým vláken a bude se provádět paralelně (každé vlákno dostane určité iterace, které vykoná). Direktiva se může uvést direktivou `parallel`, tím se provedou obě tyto direktivy najednou (programátor ušetří pár řádků kódu).

3.1.1.3 Critical

Tato direktiva říká, že následující blok kódu se bude provádět pouze jedním vláknem v daném čase (jedná se o kritickou sekci). Direktiva může mít parametr jméno kritické sekce. Všechny kritické sekce se stejným jménem jsou přípustné pouze jedním vláknem v daném čase.

3.1.1.4 Sections

Tato direktiva říká, že následující podbloky kódu (obaleny direktivou `section`) se mohou provádět paralelně aktuálním týmem vláken (opět může být uvozena direktivou `parallel` jako tomu je u direktivy `for`). Každý podblok se bude provádět právě jedním vláknem z toho týmu.

3.1.1.5 Master

Tato direktiva říká, že následující blok kódu se bude provádět pouze jednou a to hlavním vláknem (zpravidla se jedná o vlákno s id 0).

3.1.1.6 Task

Tato direktiva říká, že následující blok kódu je novým úkolem, který může být proveden nějakým vláknem. Používá se v případech, kdy direktivy `for` a `sections` jsou velmi těžkopádné (např. při rekurzi s větvením).

3.1.1.7 Taskwait

Tato direktiva říká, že následující blok kódu se bude provádět, až se vykonají v daném bloku všechny podbloky kódu uvozené direktivou `task` (čeká se na dokončení všech úkolů).

3.1.2 Klauzule

3.1.2.1 Schedule

Tato klauzule se používá u direktivy `for` a specifikuje, jaký způsobem budou jednotlivé iterace `for`-cyklu rozděleny mezi vlákna. Klauzule přijímá parametr typ rozdělování a nepovinný parametr, který určuje po jak velkých částech (kolika iteracích) se rozděluje (tento parametr je defaultně roven podílu počtu iterací a vláken). Dále jsou popsány pouze typy rozdělování `static` a `dynamic` (dalšími typy jsou `guided`, `auto` a `runtime`).

Static Celý `for`-cyklus, respektive jeho iterace se rozdělí do bloků, které jsou velké jako druhý parametr, a každé vlákno pak dostane přidělený interval těchto bloků s iteracemi, které bude provádět.

Dynamic Každé vlákno dostane místo celého intervalu pouze jeden blok s iteracemi a ten provádí. Po provedení tohoto bloku si vezme další blok, který ještě nebyl proveden. Toto se opakuje dokud se neprovedou všechny bloky.

3.1.2.2 If

Tato klauzule se používá např. u direktiv `parallel`, `for` a `sections` a říká, zda se daná direktiva má použít nebo ne (zda má kód běžet paralelně nebo sekvenčně). Parametrem je výraz, který nabývá hodnoty `true` nebo `false`.

3.1.2.3 Private

Tato klauzule se používá např. u direktiv `parallel`, `for` a `sections` a říká, že proměnná daná parametrem (proměnná vně bloku) by měla být soukromá pro každé vlákno (každé vlákno si vytvoří vlastní lokální kopii).

3.1.2.4 Reduction

Tato klauzule se používá např. u direktiv `parallel`, `for` a `sections` a říká, že na proměnnou danou parametrem se provede redukce operací, která je oddělena dvojtečkou od parametru proměnné (např. při paralelním sčítání se toto provede pro sečtení mezivýsledků od jednotlivých vláken).

3.1.2.5 Collapse

Tato klauzule se používá u direktivy `for` při více vnořených `for`-cyklů v sobě a přijímá parametr, který určuje počet vnořených `for`-cyklů, které se mají paralelizovat. Tímto se zvýší granularita práce, kterou je možné rozdat vláknům. Využití nachází především v případech, kdy práce pro jednotlivé iterace vnějšího `for`-cyklu se liší svým objemem.

3.2 Implementace sekvenčních algoritmů

Na nejvyšší úrovni je kód a po zkompilování i výsledný program rozdělen na čtyři části, respektive na čtyři programy. Těmito programy jsou `sles` (řešič pro systémy lineárních rovnic), `sleg` (generátor pro systémy lineárních rovnic), `slig` (řešič pro systémy lineárních nerovnic) a `slis` (generátor pro systémy lineárních nerovnic). Generátory byly vytvořeny pouze z důvodu testování řešičů a protože nejsou předmětem této práce, nebudou dále již zmíněny.

3.2.1 Řešení soustav lineárních rovnic

3.2.1.1 Vstup a výstup

Program `sles` na vstupu přijímá po řadě parametry formát matice A, relativní cesty k matici A a vektoru b, hodnotu ϵ pro ukončení algoritmu a vrácení výsledků a relativní cestu k adresáři, do kterého se má uložit výsledek. Výstupem je vektor x, který se uloží do souboru `x.out` do adresáře daného parametrem.

Akceptovatelné formáty pro uložení matice a vektoru odpovídají 2.3.1. Jediným rozdílem je, že soubory navíc musí obsahovat na první řádce informaci o počtu řádků, pro matice dále o počtu sloupců a nakonec pro kompresované řádky a symetrické kompresované řádky o počtu nenulových prvků. Data ze souborů jsou načtena do interních struktur, které odpovídají matici a vektoru.

3.2.1.2 Šablonovací systém

Všechny dále zmíněné struktury pro matici a vektor jsou implementovány jako šablony, jejichž parametrem je datový typ pro samotná data. Tyto šablony tak umožňují v jednotlivých programech nastavit datový typ, který bude použit, jež může mít přímý vliv na dobu výpočtu (defaultně je ve všech programech použit typ `double`, ale pokud by byl použit jiný typ s méně bity, jako např. `float`, operace na nejnižší úrovni by trvaly kratší dobu, ale za cenu snížení přesnosti výsledného řešení - tento příklad je nutno brát s rezervou, neboť záleží na dalších faktorech, jako např. na konkrétním hardwaru a jaké typy nativně implementuje). Dokonce je tak možné přijít s vlastním optimalizovaným datovým typem, který implementuje vybrané operace, a použít ho za účelem zvýšení rychlosti běhu algoritmu nebo pro zvýšení přesnosti (třída

Fraction, která uchovává reálné číslo v podobě zlomků, je ukázkou možného použití - tento typ slouží pouze pro ukázku a reálně není použitelný).

3.2.1.3 Interní struktura pro vektor

Interní strukturou pro uložení vektoru je třída Vector, která obaluje třídu vector z knihovny STL (samotná data jsou uložená v této třídě vector a navíc jsou zde implementovány metody, které jsou dále potřeba a nenalézají se ve třídě vector). Tato třída kromě metod pro načtení vektoru ze souboru a zápis vektoru do souboru obsahuje hlavně metody přetěžující operátory, které umožňují např. sečtení dvou vektorů po složkách elegantně pomocí operátoru plus (toto umožňuje mít kód hlavního programu v podobě, jak je zachycen v alg. 2.1).

3.2.1.4 Interní struktury pro matici

Interní strukturou pro uložení matice je abstraktní třída Matrix. Abstraktní je z toho důvodu, aby nebylo možné od ní vytvářet instance. Tato třída slouží jako předpis, jak konkrétní třídy pro jednotlivé formáty matice musí vypadat, aby mohly být použity v programu (umožňuje to v hlavním programu používat matici bez znalosti jejího formátu). Tato třída kromě atributů pro počet řádků a počet sloupců (každá matice musí mít počet řádků a počet sloupců) obsahuje statickou metodu getMatrixByFormat, která na základě relativní cesty k souboru a formátu vrátí matici konkrétní třídy danou formátem, a čistě virtuální metodu, která přetěžuje operátor násobení pro matici a vektor (toto je jediná metoda, kterou je nutné implementovat, neboť je potřeba násobení matice s vektorem v hlavním programu). Konkrétními třídami, které dědí od této abstraktní třídy jsou třídy MatrixF (reprezentuje matici uloženou v plném formátu), MatrixC (reprezentuje matici uloženou pomocí kompresovaných řádků) a MatrixS (reprezentuje matici uloženou pomocí symetrických kompresovaných řádků).

Třídy MatrixF, MatrixC a MatrixS se poté liší svými implementacemi načítání dat ze souboru a jejich ukládání do souboru a implementací virtuální metody pro násobení matice s vektorem, kterou jsou nuceni implementovat (dále se liší v různých metodách, které navíc obsahují, ale tyto metody se používají pouze v generátorech nebo nakonec nebyly vůbec použity). Tyto rozdíly jsou převážně mezi třídami MatrixF a MatrixC, neboť třída MatrixS dědí od třídy MatrixC s tím, že přepisuje metodu pro násobení matice s vektorem, neboť ta je odlišná (načítání, ukládání a spoustu dalšího mají společného). Třídy MatrixF a MatrixC se liší v reprezentaci dat, kde třída MatrixF uchovává data v dvojrozměrném vektoru třídy vector a třída MatrixC (tedy i třída MatrixS) ve třech vektorech třídy vector (viz. 2.3.1.2).

3.2.2 Řešení soustav lineárních nerovnic

3.2.2.1 Vstup a výstup

Program `slis` na vstupu přijímá po řadě relativní cesty k matici A, vektoru b, vektoru c a vektoru m a relativní cestu k adresáři, do kterého se má uložit výsledek. Vektor m je vektor, který pro každou proměnnou výsledného vektoru x obsahuje nulu, pokud se jedná o reálný typ, pokud se jedná o celočíselný typ, obsahuje hodnotu jedna. Předpokládá se, že matice A bude uložena v plném formátu. Výstupem je vektor x, který se uloží do souboru `x.out` do adresáře daného parametrem.

3.2.2.2 Interní struktura pro simplexovou tabulku

Interní strukturou pro simplexovou tabulku (lze vidět na obr. 1.7) je třída `SimplexTableau`. Tato třída, podobně jako třídy definované pro soustavu lineární rovnic, je implementována jako šablona (viz. 3.2.1.2) a dědí od třídy `MatrixF` (simplexová tabulka je v podstatě plná matice rozšířená o další atributy a metody).

Třída obsahuje konstruktor, který na základě vstupních parametrů matice A, vektoru b, vektoru c a vektoru m vytvoří počáteční simplexovou tabulku. Dále obsahuje kopírující konstruktor, který je potřeba v hlavním programu při větvení (při větvení se původní simplexová tabulka zkopíruje a do původní se přidá omezení typu „$\leq$“ a do kopie omezení typu „$\geq$“).

Mezi další důležité metody patří metoda `solveBySimplex`, která umí vyřešit tabulku pomocí simplexové metody, a metoda `solveByDualSimplex`, která umí vyřešit aktuální tabulku pomocí duální simplexové metody, metoda `updateTableauByKeyElement`, která upraví tabulku dle klíčového prvku v jedné iteraci simplexového algoritmu, respektive duálního simplexového algoritmu (jedná se o přičítání klíčového řádku k ostatním řádkům tak, aby v klíčovém sloupci vznikly všude nuly, kromě klíčového řádku, kde bude v tomto sloupci hodnota jedna).

Nakonec obsahuje metody `getOptimum`, `getResult`, `addSimpleConstraint` a `getRealValueRowIndex`, kde metoda `getOptimum`, vrací aktuální hodnotu optima simplexové tabulky, metoda `getResult` projde bázový vektor a sestaví výsledné řešení, metoda `addSimpleConstraint` umožňuje přidat nový řádek do simplexové tabulky s novým omezením (např. $x_1 \leq 5$ nebo $x_1 \geq 6$) a metoda `getRealValueRowIndex` vrací index řádku, kde celočíselná proměnná nabývá největší desetinné části (jedná se o řádek, na základě kterého se budou přidávat nová omezení).

3.2.2.3 Prohledávání stavového prostoru

Hlavní program obsahuje funkci `solve`, která dostane parametrem simplexovou tabulku `simplexTableau` uloženou jako instanci třídy `SimplexTableau`,

tuto tabulku rekurzivně řeší a do globálních proměnných uloží řešení a optimum (tato funkce reprezentuje jeden uzel při řešení metodou větví a mezí).

Na začátku této funkce se zavolá `solveBySimplex` nebo, pokud se nejedná o první iteraci, `solveByDualSimplex`. Pokud výsledkem je, že řešení je nepřípustné nebo neomezené, funkce končí, v opačném případu bylo nalezené optimum. Pokud toto optimum (dáno metodou `getOptimum`) je horší než globální nalezené optimum (menší, neboť se maximalizuje), funkce opět končí, v opačném případu se zkontroluje, zda všechny proměnné, které mají být celočíselné jsou celočíselné (pokud metoda `getRealValueRowIndex` vrátí nulu, vše je v pořádku, jinak se řádek, který tato metoda vrátí použije na větvení). Pokud je tato podmínka splněna, aktualizuje se globální optimum a uloží se aktuální řešení, v opačném případu je potřeba větvit dle nalezeného indexu řádku. Pokud je potřeba větvit, `simplexTableau` se zkopíruje do proměnné `simplexTableauCopy`, do původní tabulky `simplexTableau` se přidá omezení typu „$\leq$“ zavoláním `addSimpleConstraint(realValueRowIndex, true)` a do zkopírované tabulky `simplexTableauCopy` se přidá omezení typu „$\geq$“ zavoláním `addSimpleConstraint(realValueRowIndex, false)`, poté se funkce `solve` rekurzivně zavolá nad oběma tabulkami.

Po doběhnutí funkce `solve` nad tabulkou danou vstupními parametry programu obsahuje globální proměnná `globalResult` nalezené řešení a globální proměnná `globalOptimum` optimum (cenu tohoto řešení). Pokud řešení neexistuje (prostor byl nepřípustný nebo neomezený), proměnná `globalResult` bude mít velikost rovnou nule.

3.3 Implementace paralelních algoritmů

3.3.1 Řešení soustav lineárních rovnic

3.3.1.1 Paralelizace vstupu

První otázkou při paralelizaci sekvenčního algoritmu pro řešení soustav lineárních rovnic (viz. 3.2.1) je, zda je možné paralelizovat načítání matice A a vektoru b ze souborů. Odpověď je ano, lze, ale není to tak přímočaré, jak by se zdálo.

Problém je, že vlákna sdílí jedno I/O zařízení (pevný disk), které má své limity (I/O operací ze sekundu), a že většina pevných disků jsou stále plotnové, kde je potřeba nějaký čas na nastavení čtecí hlavy na místo čtení. Pokud čtení je prováděno sekvenčně, nastaví se jednou hlava a dále se čtou sekvenčně data (předpokládá se, že soubor bude sekvenčně zapsán na disk), pokud bude čtení prováděno paralelně, jednotlivá vlákna se budou střídat u tohoto I/O zařízení a téměř pokaždé se bude muset opětovně nastavovat čtecí hlava, což bude mít za následek zpomalení celého čtení souboru oproti sekvenčnímu čtení. Pro SSD disky nebo nějaký typ RAID zařízení by k zrychlení dojít mělo (záleží na konkrétní situaci).

Tento problém je řešen tak, že se nejprve celý soubor sekvenčně binárně načte do pomocného `char` pole (nabufferuje se). Dále se toto pole roztrhá podle řádků a poté již může každé vlákno zpracovávat přidělené bloky řádků. Protože každý řádek je nutno celý projít a načíst z něho dílčí data (parsování), jež zpravidla s kontrolou, zda je daná hodnota používaného datového typu (např. `double`), a s uložením do vnitřní struktury trvá oproti samotnému binárnímu načtení dat ze souboru poměrně dlouho, dochází tak k požadovanému zrychlení.

3.3.1.2 Úrovně paralelizace

Dále, jak bylo zmíněno v 2.4.2.1, je potřeba zapínat různé paralelizace až od různé velikosti vstupních instancí (platí i pro načítání ze souboru). Proto je použit parametr `VECTOR_PARALLELISM_ENABLED`, který zapíná paralelizace na úrovni vektorů, pokud je nastaven na hodnotu `true`. Paralelizace pro matice je zapnutá stále neboli se využije množství vláken, které je regulováno uživatelem vně programu (např. pro unixové systémy příkazem `export OMP_NUM_THREADS=8`).

3.3.1.3 Paralelizace operací s vektorem

Pro paralelizaci třídy `Vector` se u každé vektorové operace (operace mezi dvěma vektory nebo vektorem a skalárem) použijí direktivy `parallel` a `for` s klauzulí `schedule(static)` a `if(VECTOR_PARALLELISM_ENABLED)`. Pro skalární součin je navíc potřeba rozšířit direktivu o klauzuli `reduction(+:sum)`, která zařídí redukci do proměnné `sum` (posčítá do této proměnné mezivýsledky od jednotlivých vláken).

3.3.1.4 Paralelizace operací s maticemi

Pro paralelizace operací s maticemi, je potřeba provést pouze paralelizaci násobení matice s vektorem, jež, jak bylo zmíněno, je pro každý formát uložení matice jiná. U všech formátů se před `for`-cyklus, který je nejvíc vně přidají direktivy `parallel` a `for` s klauzulí `schedule`, kde typ rozdělování je pro každý formát jiný.

Plná matice Při násobení matice s vektorem při uložení matice jako plné by se mohla přidat klauzule `collapse(n)`, ale není ji potřeba (neboť každý řádek má stejně sloupců a tudíž je práce pro každý řádek stejně velká), dokonce se nesmí použít (pro jeden řádek r se provádí skalární součin s vektorem b, který je zapsán do výsledného vektoru na pozici r, jež by při klauzuli `collapse(2)` byla sdílena více vlákny a bylo by potřeba zamykat buňku odpovídající pozici r nebo by mohlo dojít k neočekávanému výsledku). Pro lepší představu je kód této metody zachycen v alg. 3.1.

Algoritmus 3.1 Násobení matice s vektorem - plná matice

```
function OPERATOR*(Vector<double> b)
    Vector<double> product(this → rows);

    #pragma omp parallel for schedule(static)
    for (int r = 0; r < this → rows; r++) {
        for (int c = 0; c < this → cols; c++) {
            product[r] += this → data[r][c] * b[c];
        }
    }

    return product
```

Kompresované řádky Pro uložení matice pomocí kompresovaných řádků, jak již bylo zmíněno v 2.4.2.1, je potřeba dynamické rozdělování práce (pokud by byly nenulové prvky rovnoměrně rozloženy po matici, mohlo by se použít opět statické rozdělení). Toho se docílí výměnou klíčového slova `static` za `dynamic` v klauzuli `schedule`. Navíc se do této klauzule přidá parametr `CHUNK_SIZE` (určuje, po kolika řádcích se práce rozdává mezi vlákna). Kód metody je zachycen v alg. 3.2 (proměnná `values` reprezentuje řádek s nenulovými hodnotami, proměnná `ci` reprezentuje řádek se sloupcovými indexy a proměnná `adr` reprezentuje řádek s adresami do předchozích řádků pro jednotlivé řádky).

Algoritmus 3.2 Násobení matice s vektorem - kompresované řádky

```
function OPERATOR*(Vector<double> b)
    Vector<double> product(this → rows);

    #pragma omp parallel for schedule(dynamic,CHUNK_SIZE)
    for (int r = 0; r < this → rows; r++) {
        int rowEnd = r < this → rows - 1 ? :
            this → adr[r + 1] :
            this → ci.size();

        for (int i = this → adr[r]; i < rowEnd; i++) {
            product[r] += this → values[i] * b[this → ci[i]];
        }
    }

    return product
```

Symetrické kompresované řádky Nakonec pro uložení matice pomocí symetrických kompresovaných řádků je potřeba předchozí kód rozšířit o pomocné vektory, do kterých se ukládají mezivýsledky, neboť hodnota na řádku r a sloupci c (která se přímo přečte z dat) odpovídá hodnotě na řádku c a sloupci r (která není uložená v datech). Mezivýsledky je nakonec potřeba zredukovat (nedochází k paralelní redukci, neboť redukci pro jeden řádek provádí vždy jen jedno vlákno). Tato redukce již může být prováděna se statickým rozdělením, neboť práce pro každý řádek je stejné množství (počet sloupců je roven počtu vláken). Výsledný kód této metody je zachycen v alg. 3.3 (funkce `omp_get_max_threads` vrací aktuální použitelný počet vláken a funkce `omp_get_thread_num` vrací id aktuálního vlákna).

3.3.2 Řešení soustav lineárních nerovnic

3.3.2.1 Paralelizace prohledávání stavového prostoru

Po paralelizaci načítání vstupů, jež byla řešena pro soustavu lineárních nerovnic (třída `SimplexTableau` dědí od třídy `MatrixF`, pro kterou byla paralelizace načítání matice ze souboru rozebrána v 3.3.1.1), jedinou paralelizaci, kterou je potřeba provést, je již zmíněna paralelizace prohledávání stavového prostoru. S využitím knihovny OpenMP lze toto provést třemi různými způsoby, s interním zásobníkem práce, s externím zásobníkem práce a s externími zásobníky práce s přeposíláním zpráv.

Interní zásobník První způsob, s interním zásobníkem práce, je ze všech tří způsobů nejjednodušší. Jedná se o způsob, kdy veškeré rozdělování práce se provádí interně. V hlavním programu se po načtení počáteční simplexové tabulky před zavoláním funkce `solve` přidají direktivy `parallel` a `master` (samotná funkce `solve` bude vykonána hlavním vláknem).

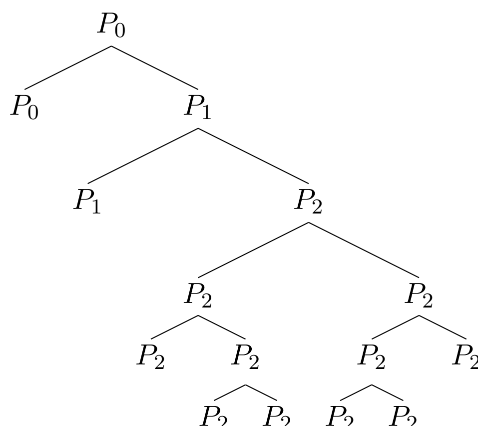
Uvnitř funkce `solve` se veškerá práce s globálním řešením `globalResult` a jeho optimem `globalOptimum` uzavře do kritické sekce `globalupdate`. Dále se před volání funkcí `solve` při větvení přidají direktivy `task` (obě dvě větve mohou běžet paralelně). Tento klasický způsob, který knihovna OpenMP umožňuje, je opravdu jednoduchý, ale jsou zde problémy, které možná nejsou na první pohled vidět.

První problém je, že dochází k častému přidávání a odebírání z interního zásobníku práce (místo, kam se po rozvětvení přidaly nové práce), jež musí být nějakým způsobem uzavřeno v kritické sekci, aby více vláken nevykonávalo tu samou práci. Tento problém by se mohl vyřešit např. tím, že vždy až po k iteracích by se použila paralelizace na větve, jinak by obě větve řešilo jedno vlákno. Bohužel by to nevedlo k dobrému přerozdělování práce a tudíž by některá vlákna většinu času nepracovala.

Druhý problém, dokonce závažnější, spočívá v tom, jak samotný algoritmus při paralelizaci vlastně probíhá. Jednu větev si ponechá samotné vlákno a

Algoritmus 3.3 Násobení matice s vektorem - symetrické kompresované řádky

```
function OPERATOR*(Vector<double> b)
    Vector<double> product(this → rows);
    Vector<Vector<double> > subProducts(
        omp_get_max_threads(),
        Vector<double>(this → rows)
    );

    #pragma omp parallel for schedule(dynamic,CHUNK_SIZE)
    for (int r = 0; r < this → rows; r++) {
        int rowEnd = r < this → rows - 1 ? :
            this → adr[r + 1] :
            this → ci.size();

        for (int i = this → adr[r]; i < rowEnd; i++) {
            subProducts[omp_get_thread_num()][r] +=
                this → values[i] * b[this → ci[i]];

            if (this → ci[i] != r) {
                subProducts[omp_get_thread_num()][this → ci[i]] +=
                    this → values[i] * b[r];
            }
        }
    }

    #pragma omp parallel for schedule(static)
    for (int r = 0; r < this → rows; r++) {
        for (int c = 0; c < omp_get_max_threads(); c++) {
            product[r] += subProducts[c][r];
        }
    }

    return product
```



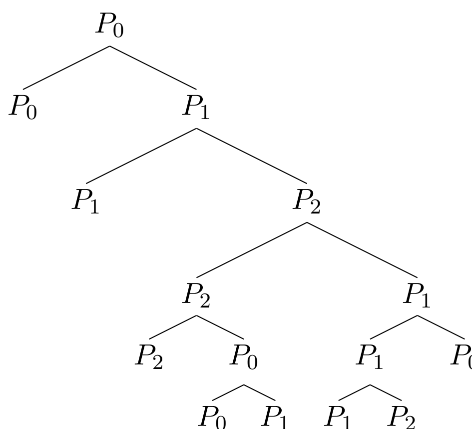
Obrázek 3.1: Prohledávání stavové prostoru - čekání

druhá se přidá do zásobníku práce, kterou si v nejpravděpodobnějším případu odebere jiné vlákno. Pokud to jiné vlákno stihne vyřešit druhou větev dříve, všechno je v pořádku, ale pokud ne, vlákno, které si vzalo první větev a je vláknem pro daný uzel, musí čekat na dokončení druhé větve, než vrátí výsledek. Toto má za následek, že po nějaké době mohou všechna vlákna kromě jednoho čekat ve větvi a jejích podvětvích, kde čekají než zbylé vlákno dokončí svoji práci. Schéma možného čekání lze vidět na obr. 3.1.

Externí zásobník Druhý způsob, s externím zásobníkem práce, řeší problémy prvního způsobu. V hlavním programu uvnitř direktivy `parallel` pod zavoláním funkce `solve` nad počáteční simplexovou tabulkou, která je vykonána hlavním vláknem, se přidá nekonečná smyčka, do které vstoupí všechna ostatní vlákna a po dokonání práce hlavního vlákna i samotné hlavní vlákno.

Na začátku se vlákno, které do této smyčky vstoupí, podívá to externího zásobníku práce (globální proměnná `taskPool`), jestli nějakou obsahuje a pokud ano, tak je mu přidělena. Tato část je v kritické sekci `taskpool`. Dále pokud vlákno dostalo práci, tak zvýší počet pracujících vláken o jedna, jež je reprezentováno proměnnou `threadsWorkingNumber`, a zavolá funkci `solve` nad přidělenou simplexovou tabulkou. Až se vynoří z této funkce, sníží počet pracujících vláken o jedna (inkrementace a dekrementace této proměnné je uzavřeno v kritické sekci `threadsWorkingNumber`). Pokud v dané iteraci smyčky nedostane práci, podívá se, zda proměnná `threadsWorkingNumber` je větší než nula (zda některé vlákno ještě pracuje) a pokud ano, pokračuje do další iterace této smyčky, pokud ne, opustí tuto smyčku a tím jeho práce končí.

Uvnitř funkce `solve` pak dochází ke změně pouze při větvení. Místo původních direktiv `task` pro obě dvě větve, se vlákno podívá do externího zásobníku práce a pokud tam je nedostatek práce (počet simplexových tabulek, které je potřeba vyřešit, je menší než parametr `TASKPOOL_MAX_SIZE`), přidá jednu vě-



Obrázek 3.2: Prohledávání stavové prostoru - častá komunikace

tev do externího zásobníku práce (v kritické sekci `taskpool`) a druhou sám vyřeší. Pokud je práce dostatek, vyřeší obě větve sám.

Díky tomuto způsobu nenastane, že by nějaké vlákno nedostalo práci, pokud nějaká práce existuje, a dále vlákno nemusí čekat na dokončení druhé podvětve a může tak dál vykonávat užitečnou práci. Navíc po počátečním rozjezdu (než všechna vlákna dostanou práci) a zaplnění externího zásobníku práce, každé vlákno řeší obě dvě své větve a nedochází tak ke zbytečné komunikační režii, jak tomu bylo u prvního způsobu.

Bohužel ani tento způsob není moc dobrý. Kód obsahuje poměrně dost kritických sekcí a stále společný zásobník práce. Pokud podvětve budou málo hluboké, bude docházet k častému přerozdělování práce pomocí externího zásobníku práce a k častému uzavírání do kritických sekcí, což způsobí výrazné zpomalení, neboť vlákna si budou většinu času pouze přerozdělovat práci. Upravené schéma z prvního způsobu pro tento problém lze vidět na obr. 3.2 (ke komunikaci ve stromě v schématu dochází pokaždé, když se rodič liší od potomka).

Externí zásobníky s přeposíláním zpráv Třetí způsob, s externími zásobníky práce s přeposíláním zpráv, jak už název napovídá, každé vlákno bude mít vlastní zásobník práce a práci si budou přeposílat pomocí soukromých zpráv, aby nedocházelo tak k časté komunikační režii. Tento způsob nejvíce využívá myšlenky popsané v analýze a odpovídá algoritmu navrženému pro práci s distribuovanou pamětí, ale využívá toho, že se paralelizuje nad sdílenou pamětí, tudíž není potřeba přeposílat zprávy, jen v hodným způsobem nastavovat globální proměnné.

Hlavní myšlenka je taková, že každé vlákno bude mít vlastní zásobník práce, kam bude odkládat práci do rezervy. Jakmile tento zásobník zaplní, řeší obě podvětve sám. V případě potřeby tento zásobník rozdělí a rozdá práci, aby

všechna vlákna mohla pracovat. Pro minimální komunikaci musí být přidáno hned několik proměnných.

Nejprve proměnná `threadWorkingTaskPools`, která reprezentuje zásobníky práce pro jednotlivá vlákna. Maximální velikost těchto zásobníků je dána parametrem `WORKING_TASKPOOL_SIZE`. Protože tyto zásobníky jsou implementovány pomocí třídy `vector`, tak aby nedocházelo k časté realokaci, zásobníky mají pevnou velikost a aktuální velikosti zásobníků pro každé vlákno drží proměnná `threadWorkingTaskPoolSizes`. Dále matice `threadMessages` o rozměrech počet vláken krát počet vláken, do které se nastavují příznaky žádosti o práci. Hodnota `true` v řádku r a sloupci c znamená, že vlákno c žádá o práci vlákno r. Matice o stejné velikosti `threadIncomingTaskPools`, kde jednotlivými prvky jsou krátké omezené zásobníky (dáno parametrem `INCOMING_TASKPOOL_SIZE`) s příchozí prací (v řádku r a sloupci c je přeposlaná práce od vlákna r vláknu c) a proměnná `threadIncomingTaskPoolSizes` drží velikosti těchto zásobníků (stejný důvod jako `threadWorkingTaskPools`). Dále je potřeba přidat zámky pro jednotlivé buňky těchto matic, těmi jsou proměnné `threadMessageLocks` a `threadIncomingTaskPoolLocks`.

Všechny tyto proměnné zajišťují minimální komunikaci. Díky zámkům na jednotlivé buňky, není potřeba dávat do kritické sekce celé matice. Protože pro přeposílání práce tu je matice zásobníků, nedochází prakticky k žádné komunikační režii. Každá dvě vlákna mají svůj vlastní příchozí a odchozí zásobník, do kterého mohou přistoupit pouze tyto dvě vlákna. Protože jedno z vláken, pokud má dojít k předání práce, práci nemá a tudíž jenom kontroluje, zda mu nějaká práce nepřišla, druhé vlákno může kdykoliv do tohoto zásobníku přidat práci bez čekání na uvolnění kritické sekce (pokud by nastal případ, že by první vlákno zrovna práci odebíralo, tak ale druhé vlákno mu žádnou práci nepřepošle a opět nebude čekat).

Nakonec jsou potřeba proměnné pro ukončení paralelního výpočtu, které bylo podrobně popsáno na začátku podsekce 2.4.2 (tento způsob je zvolen, aby bylo co nejméně kritických sekcí). Těmito proměnnými jsou proměnná `tokenRingColor`, která reprezentuje barvu peška, `tokenRingPosition`, která říká, u kterého vlákna pešek je, proměnná `threadColors`, která nese informaci o barvě každého vlákna a proměnná `canFinish`, jež je signálem, že pešek oběhl kolo a je bílý, tudíž se může ukončit výpočet.

V hlavním programu uvnitř bloku ohraničeného direktivou `parallel`, podobně jako ve druhém způsobu, je po bloku, který vykonává pouze hlavní vlákno, smyčka. Tato smyčka není nekonečná, ale vlákno do ní vstoupí pouze, pokud proměnná `canFinish` nabývá hodnoty `false` (paralelní výpočet ještě nebyl ukončen). Na začátku smyčky vlákno opět potřebuje získat práci, tak se podívá do proměnné `threadWorkingTaskPoolSizes` do svého zásobníku práce (lze vidět, že není potřeba žádná kritická sekce). Pokud práci má, jde jí vykonávat s tím, že dekrementuje velikost svého zásobníku, respektive proměnnou `threadWorkingTaskPoolSizes` pro svůj zásobník, pokud práci nemá, podívá se do proměnné `threadIncomingTaskPools` do svého sloupce a sesbírá

všechnu práci, která mu byla přeposlána. Pokud nějakou práci takto získal, nastaví příznaky ve svém sloupci v proměnné `threadMessages` na `false`, aby mu už žádné vlákno prozatím práci neposílalo, pokud žádnou práci nesesbíral nastaví tuto proměnnou ve svém sloupci na hodnotu `true` a dále v kritické sekci s názvem `tokenRingPosition` zkontroluje podmínky pro ukončení paralelního výpočtu.

Uvnitř funkce `solve` jednou za `MESSAGE_CHECK_ITER_NUM` iterací se vlákno podívá do svého řádku do proměnné `threadMessages`, zda mu přišly nějaké žádosti. Všechny žádosti (každý sloupec, který byl nastaven na hodnotu `true`), nastaví na hodnotu `false`, neboť se danými žádostmi bude zabývat. Z těchto požadavků vybere ještě ty, které v aktuální moment mají prázdné hlavní zásobníky (`threadWorkingTaskPoolSizes`) i zásobníky pro příchozí práci pro dané vlákno (`threadIncomingTaskPoolSizes`). Žádná z těchto částí nemusí být uzavřená v kritické sekci, neboť v nejhorším případu se stane, že vlákno v dané iteraci některému vláknu dá práci navíc, což ničemu nevadí. Ze zbývajících požadavků vypočítá koeficient dělení (velikost zásobníku dělena počtem požadavků plus jedna, neboť sám také potřebuje práci). Pokud tento koeficient je větší než parametr `WORKING_TASKPOOL_SPLIT_SIZE` práci rovnoměrně rozdělí mezi žádající vlákna, respektive jim tuto práci předá do odpovídajících sloupců proměnné `threadIncomingTaskPools`, a upraví velikost jednotlivých zásobníků.

Výhodou tohoto způsobu je, že pokud nastane případ, který byl zmíněn na konci druhého způsobu, tak sice dojde k častému přerozdělování práce (tomu se nedá vyhnout), ale protože se nejedná o jeden sdílený zásobník, tak komunikační režie bude mnohem menší (není potřeba tolik kritických sekcí). Algoritmus by se dal dále modifikovat různými děleními zásobníku nebo posíláním žádostí o práci, jak bylo zmíněno v analýze.

3.3.2.2 Paralelní alokátor

Nakonec je potřeba použít paralelní alokátor dynamické paměti, protože defaultní alokátor (stará se o alokaci paměti při volání funkcí jako jsou `malloc` nebo `new`), který C++ implementuje, je sekvenční. Paralelní kopírování simplexových tabulek při větvení trvá několikrát déle než kdyby veškeré kopírování bylo prováděno sekvenčně.

Knihovny, které umožňují paralelní alokaci dynamické paměti, jsou např. `ptmalloc`, `gperftools` (`tcmalloc`) nebo `hoard`. Knihovny stačí pouze nalinkovat a není potřeba měnit samotný kód. Vybral jsem `ptmalloc`, protože se jedná o malou knihovnu, která neobsahuje nepotřebné věci navíc. Domovská stránka pro tuto knihovnu je `malloc.de`.

Kapitola 4

Měření

Tato kapitola obsahuje grafy různých měření a komentářů, čeho je možné si všimnout z daného měření. Měření bylo prováděno nad různými instancemi vygenerovanými generátory (zmíněných v 3.2). Každá hodnota v grafu odpovídá průměru ze 100 měření se stejnými parametry z důvodu, aby se možná co nejvíce eliminovaly náhodné odchylky od normálu (reálně by bylo potřeba tisíce měření, ale to není účelem této práce). Pokud není dále řečeno jinak, doby výpočtu odpovídají časům potřebným na samotný výpočet (není započítaný čas na načtení vstupu a uložení výsledku, tyto grafy lze najít v příloze).

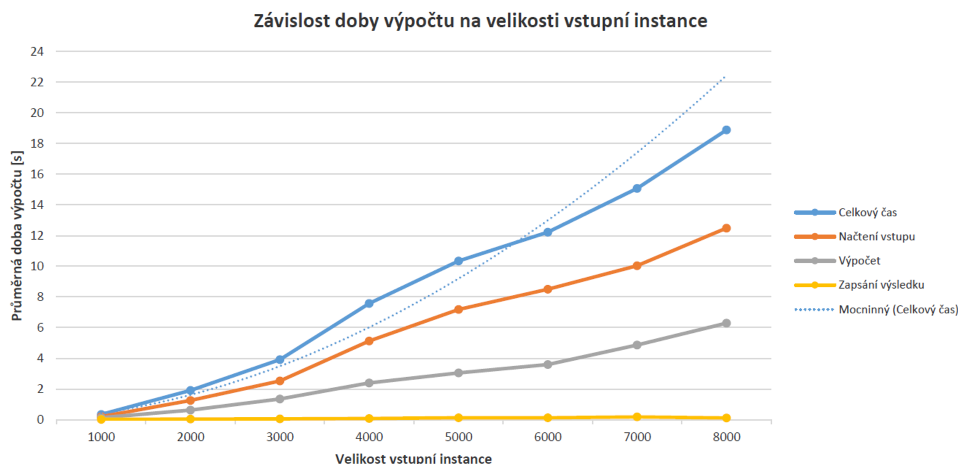
Většina měření byla prováděna na 64 bitové verzi operačního systému Microsoft Windows 10 Education v prostředí Cygwin (kolekce svobodných programů, napodobujících pod Microsoft Windows chování unixových systémů) na počítači, který obsahuje procesor Intel Core i7-4700MQ, kde jednotlivá 4 jádra s taktovací frekvencí 2.40 GHz umožňují vytvořit až 8 virtuálních procesorů, a dále obsahuje operační paměť o velikosti 16 GB.

Pro měření paralelního běhu řešení soustav lineárních nerovnic byl použit výpočetní svazek `star`, který se nachází na `star.fit.cvut.cz`. Tento svazek obsahuje jeden centrální počítač a dalších 14 výpočetních uzlů s operačním systémem Linux, konkrétně se jedná o 64 bitovou distribuci Gentoo verze 4.4.3. Každý uzel z první poloviny obsahuje 2 dvoujádrové procesory AMD Opteron 2218 s taktovací frekvencí 2.6 GHz a s operační pamětí 8 GB a každý uzel z druhé poloviny obsahuje 2 šestijádrové procesory AMD Opteron 6C Processor Model 2435 s taktovací frekvencí 2.6 GHz a s operační pamětí 26 GB.

4.1 Sekvenční algoritmy

4.1.1 Řešení soustav lineárních rovnic

Pro dobu výpočtu navrženého a implementovaného algoritmu mají po řade největší vliv velikost vstupní instance, formát uložení matice a hustota matice.

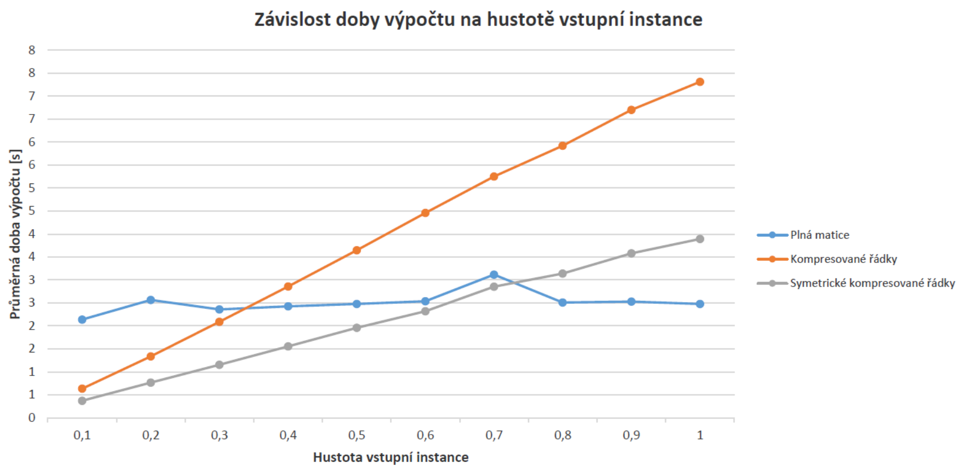


Graf 4.1: Závislost doby výpočtu na velikosti vstupní instance

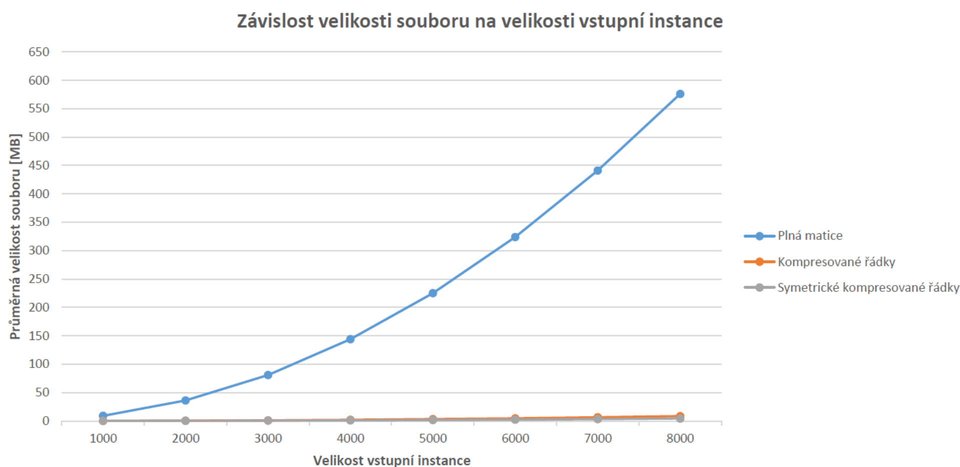
V grafu 4.1 je znázorněna závislost celkové doby výpočtu na velikosti vstupní instance pro uložení matice jako plné (jednotlivé křivky odpovídají časům potřebným na načtení vstupu, samotný výpočet a uložení výsledku). Z grafu lze vidět, že celková doba výpočtu roste téměř kvadraticky s velikostí vstupní instance, z čehož plyne, že algoritmus dobře konverguje (protože asymptotická mez sekvenčního algoritmu je $O(n^3)$). Tento graf je zde především z toho důvodu, že v některých dalších měření se již velikost vstupní instance nastaví na fixní dostatečně velkou hodnotu (pro kompresované řádky a symetrické kompresované řádky není velikost vstupní instance tolik zajímavá jako samotná hustota a tím počet nenulových prvků) a měří se jen doba samotného výpočtu.

V grafu 4.2 je znázorněna závislost doby výpočtu na hustotě nenulových prvků instance pro jednotlivé formáty uložení matice při konstantní velikosti vstupní instance nastavené na 5000. Z grafu lze vidět, že hustota téměř nemá vliv na dobu výpočtu pro uložení matice jako plné (odchylky mohou být dané konvergencí samotného algoritmu), doba výpočtu roste lineárně s hustotou vstupní matice při použití kompresovaných řádků a symetrických kompresovaných řádků, symetrické kompresované řádky jsou téměř dvakrát rychlejší a uložení pomocí kompresovaných řádků a symetrických kompresovaných řádků se vyplatí pouze do jisté hustoty.

V grafu 4.3 je znázorněna závislost velikosti souboru s maticí na disku na velikosti vstupní instance pro jednotlivé formáty uložení matice při konstantní hustotě nastavené na 0,01 (odpovídá velmi řídké matici). Z grafu lze vidět, že velikost na disku pro uložení matice jako plné roste kvadraticky s velikostí vstupní instance, pro kompresované řádky a symetrické kompresované řádky roste také kvadraticky, ale v porovnání s plnou maticí je jejich velikost zanedbatelná (pokud by se hustota zvyšovala, velikosti pro uložení matic pro



Graf 4.2: Závislost doby výpočtu na hustotě vstupní instance



Graf 4.3: Závislost velikosti souboru na velikosti vstupní instance

jednotlivé formáty by kopírovaly křivky v předchozím grafu). Tento graf slouží pro lepší představu o množství dat, které je nutné uchovávat nejen na disku, ale i v operační paměti.

4.1.2 Řešení soustav lineárních nerovnic

Pro dobu výpočtu navrženého a implementovaného algoritmu mají po řade největší vliv poměr celočíselných proměnných (od LP přes MILP až k ILP), počet těchto proměnných a počet omezení. Velkou roli také hraje topologie prohledávaného stavového prostoru, která je ale špatně měřitelná (bylo by možné měřit instance problémů, které mají přípustný prostor tvaru např. pra-

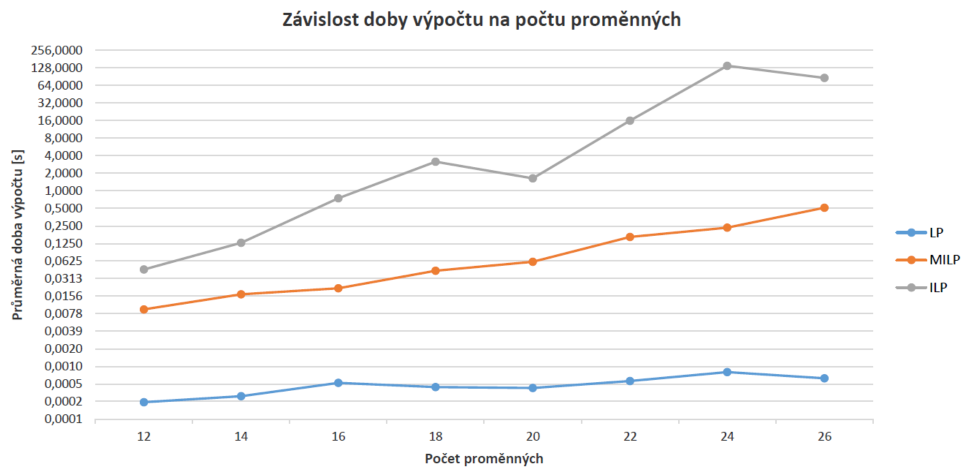
Graf 4.4: Závislost doby výpočtu na poměru celočíselných proměnných

videlného mnohostěnu, ale to je nad rámec této práce).

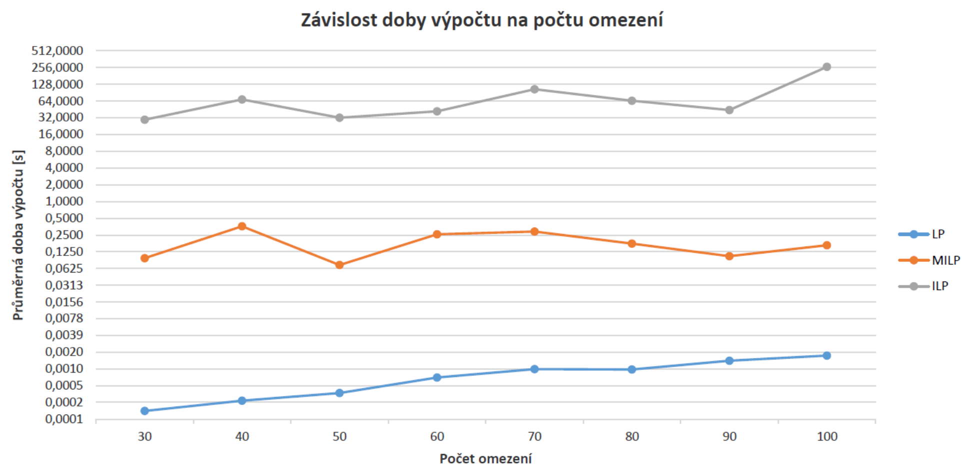
V grafu 4.4 je znázorněna závislost doby výpočtu na poměru celočíselných proměnných s konstantním počtem proměnných nastaveným na 25 a s počtem omezení na 60. Z grafu lze vidět, že křivka pro dobu výpočtu se podobá exponenciální křivce. To odpovídá tomu, že problém LP má polynomiální složitost a problém ILP má exponenciální složitost. Pro další měření se budou brát poměry celočíselných proměnných nastavené na 0, 0,5 a 1, které budou v grafech označeny jako LP, MILP a ILP.

V grafu 4.5 je znázorněna závislost doby výpočtu na počtu proměnných pro problémy LP, MILP a ILP s konstantním počtem omezení nastaveným na 60. Pro osu y je použito logaritmické měřítko se základem 2, aby bylo vidět z grafu, že problém typu MILP také roste s počtem proměnných. Dále lze z grafu vidět, že doba výpočtu pro ILP problém roste téměř exponenciálně (až na nějaké odchylky) s počtem proměnných (počet proměnných se zvyšuje o 2, ale průměrná doba výpočtu vzroste 4x, někdy i 8x).

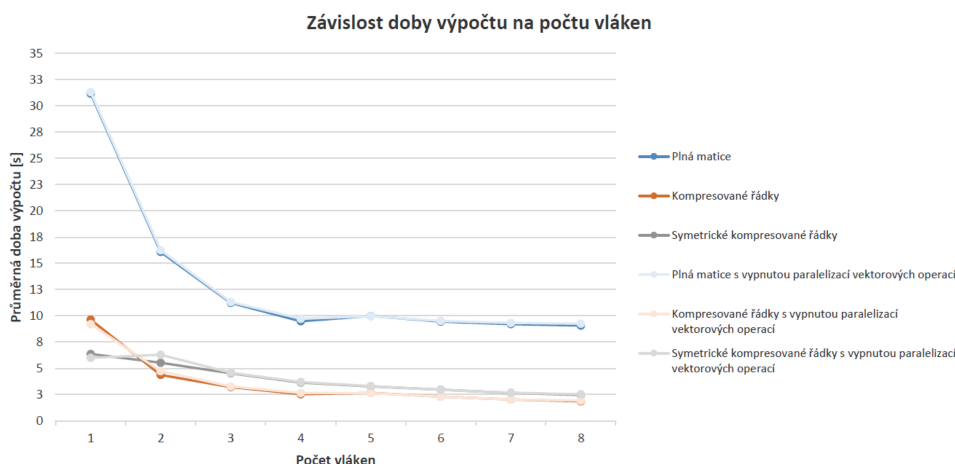
V grafu 4.6 je znázorněna závislost doby výpočtu na počtu omezení pro problémy LP, MILP a ILP s konstantním počtem proměnných nastaveným na 20. Opět je pro osu y použito logaritmické měřítko se základem 2, aby bylo vidět i pro LP a MILP, jak se mění doba výpočtu s počtem omezení. Z grafu lze vidět, že doba výpočtu příliš nezávisí na počtu omezení pro problémy MILP a ILP. To je dáno tím, že na rozdíl od problému LP zde již převládá topologie prohledávaného prostoru, která příliš nezávisí na počtu omezení. Pro problém LP se řeší pouze jedna simplexová tabulka, kde počet omezení, respektive počet řádků, které se musí přepočítat v jedné iteraci tohoto algoritmu, ovlivňuje výrazně běh algoritmu.



Graf 4.5: Závislost doby výpočtu na počtu proměnných



Graf 4.6: Závislost doby výpočtu na počtu omezení



Graf 4.7: Závislost doby výpočtu na počtu vláken

4.2 Paralelní algoritmy

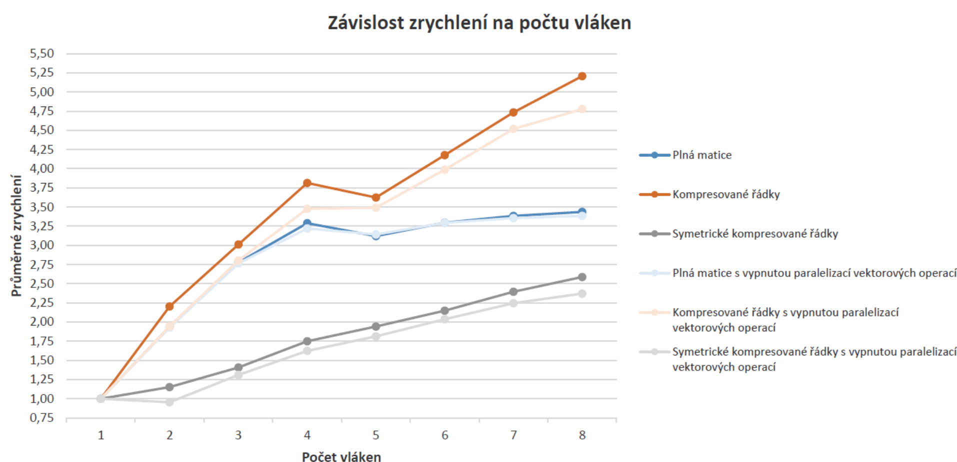
4.2.1 Řešení soustav lineárních rovnic

Pro paralelní implementaci navrženého algoritmu má dále smysl měřit závislosti doby výpočtu a zrychlení na počtu vláken.

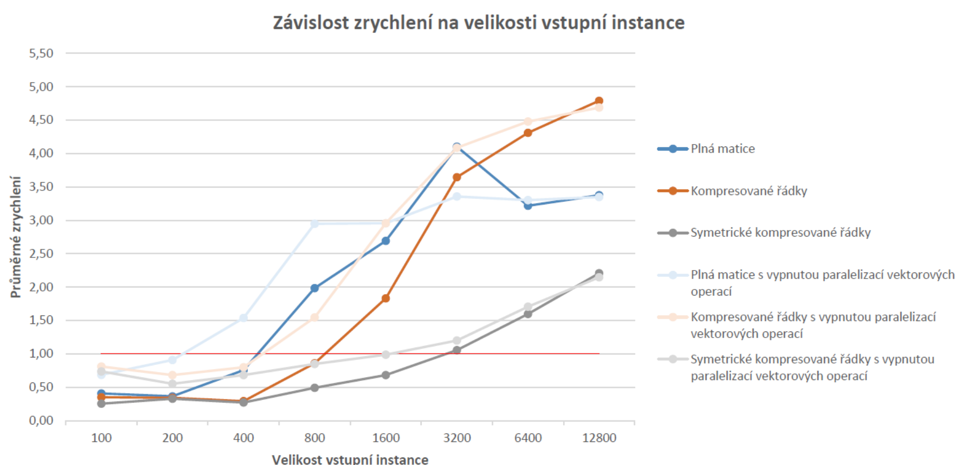
V grafu 4.7 je znázorněna závislost doby výpočtu na počtu vláken pro jednotlivé formáty uložení matice při konstantní velikosti vstupní matice nastavené na 20000 a hustotě na 0,1. Z grafu lze vidět, že doba výpočtu pozvolna klesá s rostoucím počtem vláken pro všechny formáty uložení matice. Drobný nárůst může být způsoben přerozdělováním práce uvnitř algoritmu. Dále lze vidět, že paralelizace vektorových operací nemá příliš velký vliv na dobu běhu algoritmu. To je způsobeno tím, že pro tak velké matice jako je tato, je počet operací nutných k vykonání vektorových operací v porovnání s operacemi nutných při násobení matice s vektorem zanedbatelný.

V grafu 4.8 je znázorněna závislost zrychlení výpočtu na počtu vláken pro jednotlivé formáty uložení matice při konstantních velikostí vstupní matice nastavené na 20000 a hustotě na 0,1 (jedná se o stejné měření jako v předchozím případu). Z grafu lze vidět, že nejvyššího zrychlení dosahují kompresované řádky. Symetrické kompresované řádky jsou i sekvenčně velmi rychlé a tudíž už nedosahují takového zrychlení (pro větší nebo hustší matice by bylo dosaženo většího zrychlení). Pro uložení matice jako plné se v tomto měření doba výpočtu a zrychlení stabilizuje od 4 vláken. U všech formátů dochází k nepatrnému nárůstu zrychlení při zapnuté paralelizaci vektorových operacích (oproti předchozímu grafu zde lze vidět rozdíl).

V grafu 4.9 je znázorněna závislost zrychlení výpočtu na velikosti vstupní instance pro jednotlivé formáty uložení matice při konstantním počtu vláken

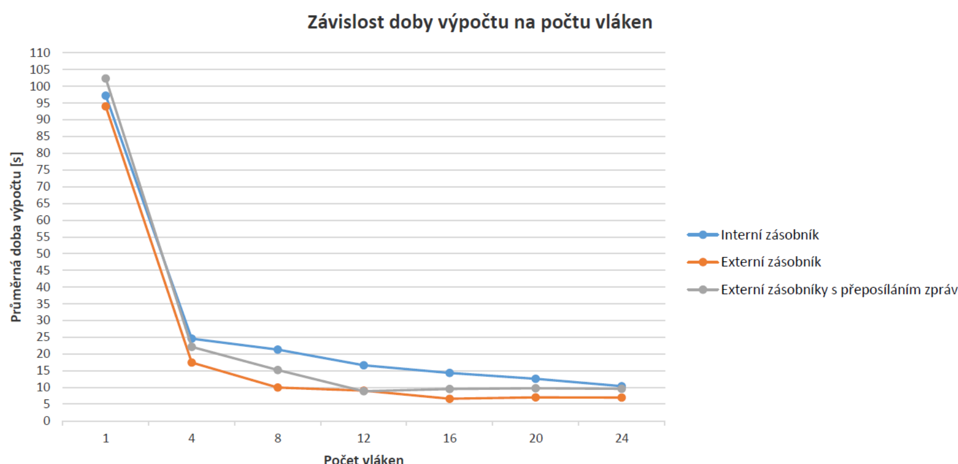


Graf 4.8: Závislost zrychlení výpočtu na počtu vláken



Graf 4.9: Závislost zrychlení výpočtu na velikosti vstupní instance

nastaveným na 8 a hustotě na 0.1. Z grafu lze vidět, že paralelizovat má smysl až od určité velikosti vstupní instance. Dále je vidět, že plná matice lze paralelizovat lépe pro malé instance než zbylé dva formáty. To je způsobeno tím, že u ní nehraje roli počet nenulových prvků, který je pro zbylé dva formáty rozhodující. Dále lze vidět, že pro všechny formáty o těchto velikostech vstupních instancích nemá smysl paralelizovat vektorové operace, protože běh algoritmu akorát zpomalují (u předchozího grafu s velikostí vstupní instance nastavené na 20000 již k nepatrnému nárůstu zrychlení s paralelizací vektorových operacích dochází).



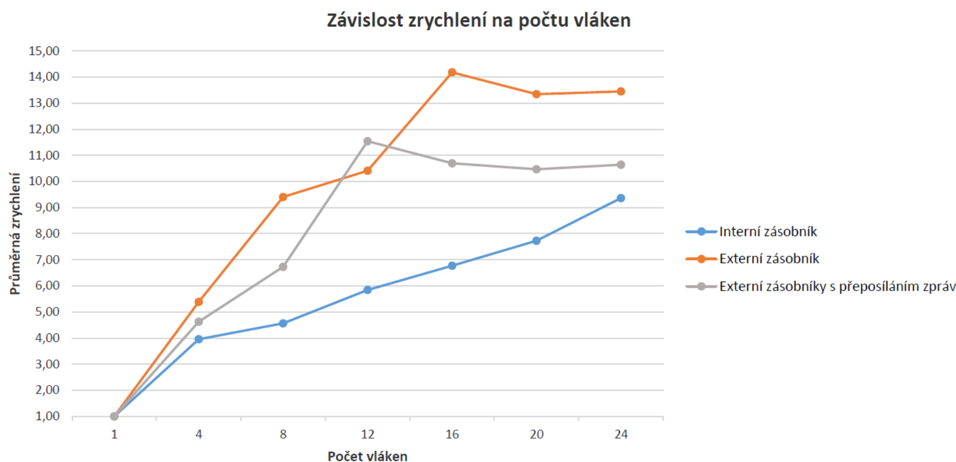
Graf 4.10: Závislost doby výpočtu na počtu vláken

4.2.2 Řešení soustav lineárních nerovnic

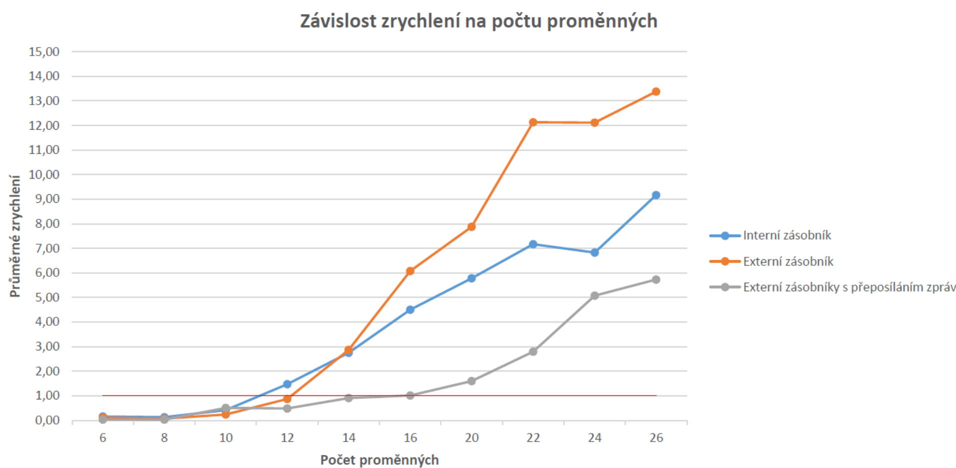
Pro paralelní implementaci navrženého algoritmu má dále smysl měřit závislosti doby výpočtu a zrychlení na počtu vláken a na typu zásobníků práce (pro připomenutí, těmito zásobníky jsou interní zásobník, externí zásobník a externí zásobníky s přeposíláním zpráv). Veškeré další měření je prováděno pro úlohy typu ILP, neboť je ze všech tří typů úloh nejsložitější a prohledávání stavového prostoru, které se paralelizuje, je touto úlohou nejvíc ovlivněno (pro druhý extrém LP se naopak nic neparalelizuje).

V grafu 4.10 je znázorněna závislost doby výpočtu na počtu vláken pro zmíněné typy zásobníků (viz. 3.3.2) s konstantním počtem proměnných nastaveným na 25 a počtem omezení na 80. Z grafu lze vidět, že překvapivě nejlépe si vede algoritmus, který implementuje externí zásobník. Toto je nejspíš způsobeno tím, že náhodně vygenerované problémy mají strukturu prohledávaného prostoru s převahou dlouhých větví, tudíž nedochází k tak častým požadavkům do společného externího zásobníku a oproti algoritmu s externími zásobníky s přeposíláním zpráv nedochází k zbytečné režii při předávání práce.

V grafu 4.11 je znázorněna závislost zrychlení výpočtu na počtu vláken pro jednotlivé typy zásobníků s konstantním počtem proměnných nastaveným na 25 a počtem omezení na 80 (jedná se o stejné měření jako v předchozím případu). Z grafu lze vidět, že k největšímu zrychlení dochází u algoritmu s externím zásobníkem, dokonce při 4 a 8 vláknech dochází k superlineárnímu zrychlení, které bylo vysvětleno v 1.4.1 (zrychlení je větší než počet vláken). Menší zrychlení oproti méně vláknům může být způsobeno častým přerozdělováním práce a také hlavně tím, že náhodně vygenerované instance často mají obrovské rozdíly v dobách výpočtů (při větším počtu testování by nemělo docházet k těmto výchylkám).

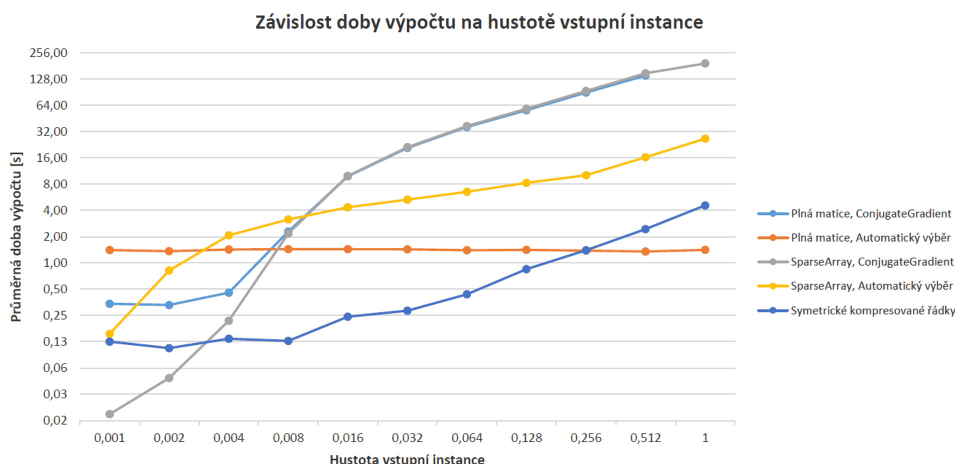


Graf 4.11: Závislost zrychlení výpočtu na počtu vláken



Graf 4.12: Závislost zrychlení výpočtu na počtu proměnných

V grafu 4.12 je znázorněna závislost zrychlení výpočtu na počtu proměnných pro jednotlivé typy zásobníků s konstantním počtem vláken nastaveným na 24 a počtem omezení na 80. Z grafu lze vidět, že paralelizovat má smysl až od určitého počtu proměnných. Dále je vidět, že pro všechny typy zásobníků průměrné zrychlení pozvolna stoupá s počtem proměnných a že algoritmus s externím zásobníkem lze paralelizovat nejlépe (už pro malý počet proměnných překročí sekvenční hranici a pak roste nejstrměji).



Graf 4.13: Závislost doby výpočtu na hustotě vstupní instance

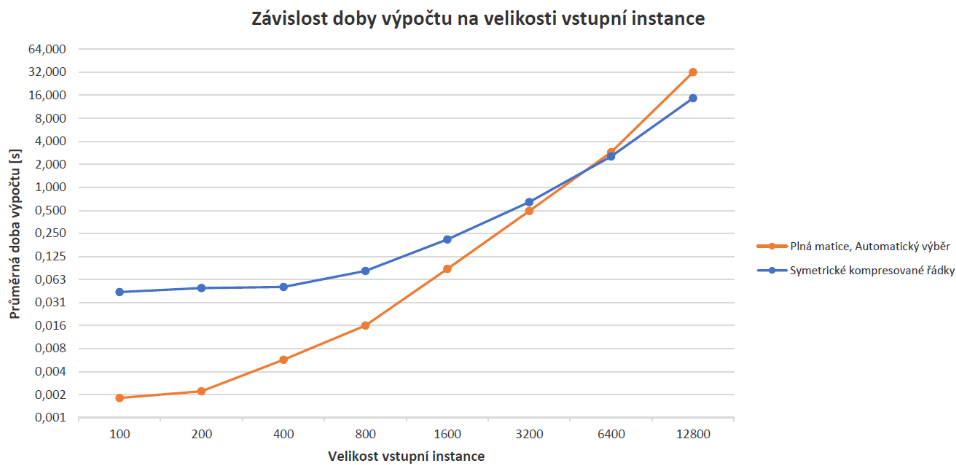
4.3 Srovnání se známými řešiči

4.3.1 Řešení soustav lineárních rovnic

Testování oproti programu Wolfram Mathematica (viz. 2.2.3.1) probíhá vytvořením skriptu obsahující data v obou popsaných formátech a volání zmíněné funkce `LinearSolve`, která je obalená funkcí `AbsoluteTiming`, pro měření doby výpočtu. Tento skript se pak dává na vstup programu wolfram při jeho pouštění z příkazové řádky.

V grafu 4.13 je znázorněna závislost doby výpočtu na hustotě nenulových prvků instance pro uložení matice jako plné nebo jako `SparseArray` s použitím metody `ConjugateGradient` nebo automatické volby. Tyto 4 možnosti (kombinace) jsou dány k porovnání s uložením matice pomocí symetrických kompresovaných řádků vlastní implementace. Test je puštěn s konstantní velikostí vstupní instance nastavené na 5000 a počtem vláken na 8. Pro osu y je použito logaritmické měřítko se základem 2, aby byly v grafu zachyceny i detailní rozdíly pro nízké hodnoty. Z grafu lze vidět, že pro nejřidší matice dosahuje metoda `ConjugateGradient` spolu s uložením matice pomocí `SparseArray` nejlepších výsledků, od hustoty okolo 0,003 do 0,256 nejlepších výsledků dosahuje vlastní implementace s využitím symetrických kompresovaných řádků a od hustoty 0,256 dosahuje nejlepších výsledků automatický výběr metody spolu s uložením matice jako plné. Dále je si možné všimnou, že pro automatický výběr s plnou matici se časová složitost s hustotou nemění a že pro metodu `ConjugateGradient` je od jisté hustoty jedno, který způsob se pro uložení matice použije (navíc pro oba způsoby je běh příliš pomalý).

Na základě předchozího grafu jsem se rozhodl dále porovnat symetrické kompresované řádky vlastní implementace s automatickým výběrem a ulože-



Graf 4.14: Závislost doby výpočtu na velikosti vstupní instance

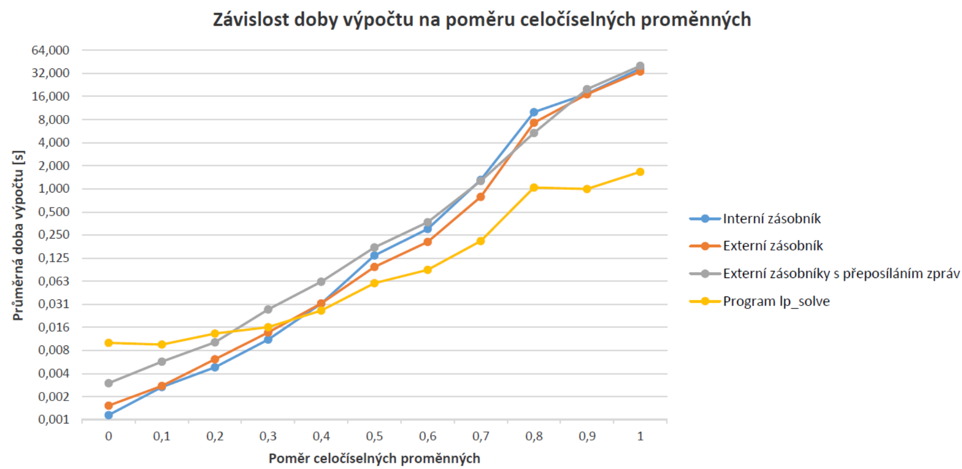
ním matice jako plné pro funkci `LinearSolve`, neboť z naměřených hodnot mi tyto dva způsoby přijdou nejlepší. V grafu 4.14 je znázorněna pro tyto způsoby závislost doby výpočtu na velikosti vstupní instance při konstantním počtu vláken nastaveným na 8 a hustotě na 0,256 (hustota, kde se v předchozím grafu protínají). Z grafu lze vidět, že od jisté velikosti vstupní instance s konstantní hustotou je výhodnější použit vlastní implementaci se symetrickými kompresovanými řádky.

4.3.2 Řešení soustav lineárních nerovnic

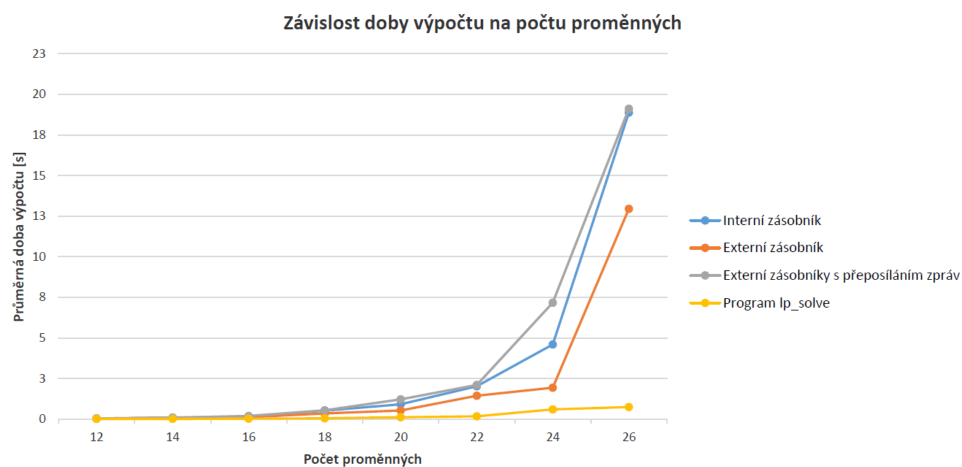
Testování oproti programu lp_solve (viz. 2.2.3.2) probíhá vytvořením souboru s odpovídajícími daty v `LP` formátu a pouštění z příkazové řádky s parametrem `time`, který měří dobu výpočtu. Pro testování jsem vybral `LP` formát, neboť mi přijde nejjednodušší a nejlépe čitelný.

V grafu 4.15 je znázorněna závislost doby výpočtu na poměru celočíselných proměnných pro jednotlivé typy zásobníků a program lp_solve s konstantním počtem vláken nastaveným na 8, počtem omezení na 80 a počtem proměnných na 25. Pro osu y je použito logaritmické měřítko se základem 2, aby byly v grafu zachyceny i detailní rozdíly pro nízké hodnoty. Z grafu lze vidět, že program lp_solve, přestože běží sekvenčně, dosahuje daleko lepších výsledků, dokonce při poměru celočíselných proměnných nastaveným na hodnotu 0,9 je zhruba 16x rychlejší. Vlastní implementace se mu mohou rovnat pouze při nízkých poměrech celočíselných proměnných.

V grafu 4.16 je znázorněna závislost doby výpočtu na počtu proměnných pro jednotlivé typy zásobníků a program lp_solve s konstantním počtem vláken nastaveným na 8, počtem omezení na 80 a poměrem celočíselných proměnných na 1 (ILP problém). Z grafu lze vidět, že propast mezi programem



Graf 4.15: Závislost doby výpočtu na poměru celočíselných proměnných



Graf 4.16: Závislost doby výpočtu na počtu proměnných

lp_solve a vlastními implementacemi se neúnosně zvětšuje s počtem proměnných.

Závěr

Cílem této diplomové práce bylo vybrání vhodných algoritmů řešících soustavy lineárních rovnic a nerovnic a tyto algoritmy implementovat. Dále rozebrat teoretické možnosti jejich paralelizace a tato paralelní řešení implementovat. Implementaci provést pomocí programovacího jazyka C++ a knihovny OpenMP. Nakonec provést měření na různé škále možných vstupů a porovnat sekvenční řešení, paralelní řešení a řešení třetích stran. Všechny tyto cíle byly splněny.

Z měření vyplývá, že řešič pro soustavy lineárních rovnic by mohl najít uplatnění při řídkých maticích uložených pomocí symetrických kompresovaných řádků, kde dosahuje nejlepších výsledků. Rychlejší oproti programu Wolfram Mathematica je v tomto případu z toho důvodu, že je přímo navržený pro tento specifický způsob uložení matice, jež program Wolfram Mathematica nepodporuje.

Dále z měření bohužel vyplývá, že řešič pro soustavy lineárních nerovnic není konkurenceschopný. Použít se dá pouze u MILP problémů, které mají nízký poměr celočíselných proměnných, respektive se jedná o dobrý LP řešič. Navíc pro problémy s vysokým poměrem celočíselných proměnných nenalézá vždy globální optimum z důvodu nepřesné aritmetiky. Program lp_solve je rychlejší, protože používá různé optimalizace pro různé vstupy, např. škálování nebo pivotáž, což má vliv na dobu běhu algoritmu a na kvalitu výsledného řešení.

Knihovna OpenMP potvrdila její klady a zápory. Opravdu stačí pouze krátké direktivy, aby se ze sekvenčního kódu stal paralelní, což umožňuje i méně zkušeným programátorům rychlou a snadnou paralelizaci jejich kódu. Na druhou stranu tato jednoduchost může být vykoupena nižším výkonem, jež jsme mohli vidět při paralelizaci prohledávání stavového prostoru u řešení soustav lineárních nerovnic. Přesto se v obou případech podařilo provést úspěšnou paralelizaci, která u řešiče lineárních nerovnic v některých případech může dosáhnout i superlineárního zrychlení.

Nakonec je potřeba říct, že v práci zmiňované a testované řešiče třetích

stran jsou opravdu skvělé a že jsou natolik optimalizované, že se jim dá jen stěží vyrovnat. Proto přínos této práce vidím především ve shrnutí možností, jak řešit soustavy lineárních rovnic a nerovnic, a problémů spojených s paralelizací implementovaných algoritmů, které je potřeba vyřešit.

Literatura

[1] Černý, M.: Lineární optimalizace a metody - Úvod. Přednáška, Fakulta informačních technologií, České vysoké učení technické v Praze, 2011.

[2] Kruis, J.: Nelineární optimalizace a numerické metody - Úvod. Přednáška, Fakulta stavební, České vysoké učení technické v Praze, 2015.

[3] Bečvář, J.: *Lineární algebra.* Matfyzpress, třetí vydání, 2005, ISBN 80-86732-57-6.

[4] Vastl, Z.: *Paralelní řešič soustav lineárních rovnic pro webmath.zcu.cz.* Bakalářská práce, Fakulta aplikovaných věd, Západočeská univerzita v Plzni, 2008.

[5] Krejsa, M.: Algoritmizace inženýrských výpočtů - Soustavy lineárních rovnic. Přednáška, Fakulta stavební, Vysoká škola báňská – Technická univerzita Ostrava, únor 2015.

[6] Olšák, P.: Lineární algebra - LU rozklad. Materiál k výuce, Fakulta elektrotechnická, České vysoké učení technické v Praze, 2010.

[7] Limpouch, J.: Lineární algebra - Superrelaxační metoda. [online], [cit. 2016-02-22]. Dostupné z: `http://kfe.fjfi.cvut.cz/~limpouch/numet/linalg/node25.html`

[8] Shewchuk, J. R.: An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Odborný článek, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, srpen 1994.

[9] Holmström, M.: Conjugate Direction Methods. [online], [cit. 2016-02-25]. Dostupné z: `http://user.it.uu.se/~matsh/opt/f8/node5.html`

[10] Algoritmy.net: Simplexová metoda. [online], [cit. 2016-02-26]. Dostupné z: `https://www.algoritmy.net/article/1416/Simplexova-metoda`

[11] Černý, M.: Lineární optimalizace a metody - Simplexový algoritmus. Přednáška, Fakulta informačních technologií, České vysoké učení technické v Praze, 2011.

[12] Kolman, P.: Metoda vnitřního bodu. Materiál k výuce, Matematicko-fyzikální fakulta, Univerzita Karlova, Praha, leden 2011.

[13] Černý, M.: Lineární optimalizace a metody - Metody vnitřního bodu. Přednáška, Fakulta informačních technologií, České vysoké učení technické v Praze, 2011.

[14] Šůcha, P.: Celočíselné lineární programování. březen 2004.

[15] Albert, S.: *Solving Mixed Integer Linear Programs Using Branch and Cut Algorithm*. Diplomová práce, Faculty of North Carolina State University, Raleigh, NC, 2006.

[16] Tvrdík, P.: Paralelní algoritmy a systémy - Složitost a škálovatelnost paralelních algoritmů. Přednáška, Fakulta informačních technologií, České vysoké učení technické v Praze, 2014.

[17] Lórencz, R.; Hlaváč, J.; Zahradnický, T.: Architektura počítačových systémů - Kvantitativní principy návrhu počítačů. Přednáška, Fakulta informačních technologií, České vysoké učení technické v Praze, 2012.

[18] Mehmood, R.; Crowcroft, J.: Parallel iterative solution method for large sparse linear equation systems. Technická zpráva, University of Cambridge, Cambridge, UK, říjen 2005.

[19] Lyu, J.; Luh, H.; Lee, M.-C.: Solving Linear Programming Problems on the Parallel Virtual Machine Environment. *American Journal of Applied Sciences*, 2004: s. 90–94.

[20] Dantzig, G. B.; Wolfe, P.: Decomposition Principle for Linear Programs. *Operations Research*, 1960: s. 101–111.

[21] Hall, J. A. J.: Towards a practical parallelisation of the simplex method. *Computational Management Science*, 2008: s. 139–170.

[22] Ralphs, T.: Parallel Branch and Cut. Odborný článek, Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA, leden 2006.

[23] Alon, N.; Megiddo, N.: Parallel Linear Programming in Fixed Dimension Almost Surely in Constant Time. *Journal of the Association for Computing Machinery*, 1994: s. 422–434.

[24] Pinar, A.; Ümit V. Çatalyürek; Aykanat, C.; aj.: Decomposing Linear Programs for Parallel Solution. *Journal of the Association for Computing Machinery*, 2005: s. 473–482.

[25] Kruis, J.: Nelineární optimalizace a numerické metody - Ukládání matic. Přednáška, Fakulta stavební, České vysoké učení technické v Praze, 2015.

[26] Tvrdík, P.: Paralelní algoritmy a systémy - Paralelní architektury a modely. Přednáška, Fakulta informačních technologií, České vysoké učení technické v Praze, 2014.

[27] Dijkstra, E. W.; Scholten, C.: Termination detection for diffusing computations. *Information Processing Letters*, ročník 11, srpen 2008: s. 1–4.

[28] Zhang, W.: Branch-and-Bound Search Algorithms and Their Computational Complexity. Odborný článek, University of Southern California, Los Angeles, CA, květen 1996.

[29] Cook, W.; Hartmann, M.: On the Complexity of Branch and Cut Methods for the Traveling Salesman Problem. *DIMACS*, ročník 1, 1990.

Příloha A

Seznam použitých zkratek

CBC COIN-OR Branch and Cut

CRCW Concurrent Read Concurrent Write

CREW Concurrent Read Exclusive Write

CTMC Continuous Time Markov Chains

EREW Exclusive Read Exclusive Write

GLPK GNU Linear Programming Kit

ILP Integer Linear Programming

LP Linear Programming

MILP Mixed Integer Linear Programming

OpenMP Open Multi-Processing

PCG Preconditioned Conjugate Gradients

PRAM Parallel Random-Access Machine

RAID Redundant Array of Inexpensive/Independent Disks

SLEG System of Linear Equitions Generator

SLES System of Linear Equitions Solver

SLIG System of Linear Inequalities Generator

SLIS System of Linear Inequalities Solver

SSD Solid State Drive

STL Standard Template Library

Příloha B

Obsah přiloženého CD

- project
  - src .... adresář se zdrojovovými kódy aplikace
  - vendors .... adresář s knihovnou ptmalloc
  - Makefile .... soubor pro kompilaci programů
  - README.txt .... informace o balíčku
- thesis
  - images .... adresář s použitými obrázky
  - csn690.bst .... norma pro použitou literaturu
  - cvut-logo-bw.pdf .... logo školy
  - DP_Ondrej_Tomas_2016.pdf .... text práce ve formátu PDF
  - DP_Ondrej_Tomas_2016.tex .... text práce ve formátu LaTeX
  - FITthesis.cls .... šablona práce
  - mybibliographyfile.bib .... použitá literatura
- README.txt .... stručný popis obsahu CD